Converting an OPC UA software development kit from Java to Delphi

Teppo Uimonen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 10.10.2016

Thesis supervisor:

D.Sc. (Tech.) Ilkka Seilonen

Thesis advisor:

M.Sc. (Tech.) Jouni Aro

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Teppo Uimonen

Title: Converting an OPC UA software development kit from Java to Delphi

Date: 10.10.2016 Language: English Number of pages: 8+56

Department of Electrical Engineering and Automation

Professorship: Intelligent products

Supervisor: D.Sc. (Tech.) Ilkka Seilonen

Advisor: M.Sc. (Tech.) Jouni Aro

OPC UA provides a secure and interoperable standard for communication between devices and information systems, but to use it effectively in a software system, an OPC UA SDK written in the implementation language is needed. To develop one for Delphi programming language, this thesis studies how Prosys OPC UA Java SDK can be converted to Delphi.

Three issues are identified in the conversion. First, Java code should be translated to Delphi. Second, an interface is needed to use a C language OPC UA dynamic-link library for low-level functionality. Third, a high-level application programming interface needs to be designed carefully since modifying it afterwards requires application developers to update their applications too. Methods were studied to solve these issues, after which an early Delphi SDK prototype was implemented using the methods, verifying their feasibility for the conversion.

In future, the methods found will be used to develop the complete Delphi SDK. However, since the prototype implements perhaps about 20% of the complete SDK, there is a realistic possibility that some issues were undetected. It was also noted that even a primitive Java-to-Delphi compiler would be useful. In addition, the thesis proposes further studies about the performance of the Delphi SDK and the APIs of the Prosys OPC (UA) SDKs.

Keywords: OPC UA, Delphi, SDK

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Teppo Uimonen

Työn nimi: Java-kielisen OPC UA -ohjelmistokehitystyökalun muuntaminen Delphille

Päivämäärä: 10.10.2016 Kieli: Englanti Sivumäärä: 8+56

Sähkötekniikan ja automaation laitos

Professuuri: Älykkäät tuotteet

Työn valvoja: TkT Ilkka Seilonen

Työn ohjaaja: DI Jouni Aro

OPC UA on määrittely laitteiden ja tietojärjestelmien välisen tiedonsiirron turvaamiseksi sekä yhteensopivuuden takaamiseksi. OPC UA SDK:ta käytetään ohjelmistokirjastona toteuttamaan tarvittava toiminnallisuus. OPC UA SDK:n kehittämiseksi Delphi-ohjelmointikielelle tässä diplomityössä tutkitaan miten Prosys OPC UA Java SDK voidaan muuntaa Delphille.

Java SDK:n muuntamiseen sisältyy kolme osiota. Ensiksi Java-kielistä koodia on käännettävä Delphille. Seuraavaksi rajapinta vaaditaan matalan tason OPC UA -toiminnallisuuden tarjoavan C-kielisen dynaamisesti linkitetyn kirjaston käyttämiseksi. Sitten käydään läpi käytäntöjä hyvän ohjelmistorajapinnan suunnittelemiseksi, sillä rajapintaan jälkeenpäin tehtävät muutokset vaativat muutoksia myös sitä käyttäviin sovelluksiin. Menetelmiä osioiden toteuttamiseksi tutkittiin, minkä jälkeen niitä käytettiin onnistuneesti Delphi SDK -prototyypin kehittämiseksi.

Esitettyjä menetelmiä tullaan käyttämään valmiin Delphi SDK:n kehittämiseksi jatkossa. Prototyyppi toteuttaa kuitenkin ehkä 20% valmiista SDK:sta, joten on mahdollista, että joitain ongelmatekijöitä jäi huomaamatta. Havaittiin myös, että alkeellinenkin Java-Delphi-kääntäjä lisäisi käännöstehokkuutta. Mahdollisia jatkotutkimusaiheita ovat Delphi SDK:n suorituskykyarviointi sekä laajamittainen ohjelmistorajapinta-analyysi Prosysin OPC (UA) -kehitystyökaluista.

Avainsanat: OPC UA, Delphi, SDK

Preface

First of all, I would like to thank Prosys for offering me the thesis possibility which allows me to graduate, finally, after somewhat too many years of studying and not studying. Evidently, Aalto University deserves credits too as they decided that in order to get the one guy out of the school it is best to set a strict deadline for the graduation. Also everybody at the office deserves thanks for creating such an easy-going and entertaining atmosphere. Fortunately, I was also granted the possibility to escape this playful environment and actually write my thesis on remote working days.

I want to thank my instructor Jouni Aro for allowing great amounts of independence in the thesis process as well as offering his specialized knowledge on the topic for the best of the thesis. Ilkka Seilonen, my supervisor, deserves thanks for offering efficient feedback and guidance from the academic perspective. I thank my friends and family for support in the writing process and for many nice moments when I was able to forget about the thesis. Finally, I would like to thank my little brother Tuukka for picking up the printed versions of this thesis while I am on a vacation.

Otaniemi, 18.9.2016

Teppo Uimonen

Contents

Abbreviations

API	Application Program Interface
COM	Component Object Model
DLL	Dynamic-link Library
DCOM	Distributed COM
GUI	Graphical User Interface
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MSDN	Microsoft Developer Network
OPC	Open Platform Communications, previously OLE for Process Control
OPC UA	OPC Unified Architecture
RTL	Run-time Library
SDK	Software Development Kit
SOA	Service-oriented Architecture
TCP	Transmission Control Protocol
UML	The Unified Modeling Language
VCL	Visual Component Library

1 Introduction

1.1 Background

OPC Unified Architecture (OPC UA) is a communication standard used most commonly between automation and information layers of an industrial enterprise to provide interoperability and security. To use OPC UA for communication between different systems, the applications of the systems need to be able to perform complex tasks that are very time-consuming to implement. As a solution, several OPC UA Software Development Kits (SDKs) that implement the required functionality have been developed and are available to be used as a software component to make development of OPC UA applications effective and reliable.

Currently there are SDKs available for ANSI C, C++, C# and Java programming languages, but many industrial information systems are developed with other languages, such as Delphi. Nowadays, in the era of Industrial Internet of Things (IIoT), industrial enterprises are seeking to connect more devices with higher layer information systems and paying more attention to security as the awareness of cybersecurity is increasing, creating demand also for an OPC UA Delphi SDK.

This demand was noted by Prosys PMS Ltd, a Finnish industrial software company that decided to develop an OPC UA SDK for Delphi. The company focuses on OPC UA related projects implementing both customer-tailored projects and developing its own products for sale. These products include an OPC UA Java SDK as well as a Delphi SDK for OPC UA's predecessor OPC Classic. Having knowledge and experience both in Delphi and developing an OPC UA SDK had a major impact on the decision to start developing the SDK in Delphi. Furthermore, the availability of OPC UA software components provides several reuse possibilities that are seen to increase the software productivity and quality [23].

First, it was decided that the new Delphi SDK will be based on the implementation of the successful Prosys OPC UA Java SDK. This can be seen as a case of an internal and opportunistic reuse of a software component, meaning that the existing component is owned by the company and was not originally designed to be reused this way [22]. Also, some parts of the code can be generated automatically using UML-to-Delphi code generation since the base class architecture of the Java SDK, including many variables and methods of the classes, has been modeled in UML. In addition, the Java SDK uses an OPC UA low-level library known as the Java

Stack which does not need to be converted, instead the Delphi SDK can use the open-source ANSI C Stack as a dynamic-link library (DLL).

1.2 Objectives

The OPC UA Delphi SDK development process can be divided into different phases. These include conversion of the Java SDK code to Delphi, creation of a rapid application interface (RAD) for Delphi, development of an XML-to-Delphi code generator to use OPC UA types, software unit testing and applying for official OPC UA compliance.

This thesis focuses on the conversion process, which is the first phase and excepted to be the most demanding. It consists of source-to-source translation of the Java SDK to Delphi as well as working with the high-level application programming interface (API) and the low-level interface with the ANSI C Stack. These issues are illustrated in figure 1. The objective is to study how Prosys OPC UA Java SDK can be converted into the OPC UA Delphi SDK at Prosys PMS Ltd and to identify any inconveniences that might question the feasibility of the project or its development practices. To achieve this goal, literature is first studied to find proper methods and tools. Secondly, certain parts of the Delphi SDK, more specifically a prototype of the client SDK, will be developed to evaluate the used methods.

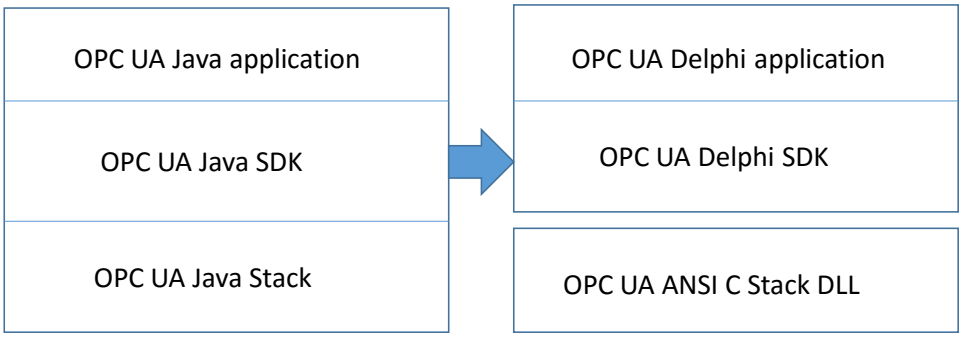

Figure 1: Identifying the conversion issues.

At the moment, it is unknown how the source code transition from Java to Delphi can succeed when it is applied to the OPC UA Java SDK. Moreover, the impacts of changing the statically linked Java Stack to the dynamically linked ANSI C Stack are not known. It is expected that these factors would influence the Delphi SDK architecture as well as create an amount of workload. At worst, the architecture

would need to be redesigned from scratch instead of using the one of the Java SDK, thus making the creation and maintenance of the Delphi SDK too laborious to be profitable. This can also be the result in case the ANSI C Stack or the Delphi framework proves to provide too little needed functionality, creating need to do more by hand.

Simultaneously, the application programming interface (API) of the Delphi SDK requires careful design already during the code conversion phase. Remarkable modifications made to the API after the product launch require users of the Delphi SDK to update their applications to be compatible with the new API when they change to a newer Delphi SDK version.

This thesis tries to answer the following research questions:

1. *How can the existing Prosys OPC UA Java SDK be translated to Delphi?*
2. *How can the wrapping of the OPC UA ANSI C Stack DLL be implemented?*
3. *How can the OPC UA Delphi SDK API be designed?*
4. *How serious and what type of disadvantages are there in the development of the Delphi SDK based on the Java SDK and on top of the OPC UA ANSI C Stack?*

1.3 Research methods

Literature is first examined to study OPC UA and the conversion methods that include the Java-to-Delphi translation, wrapping a C-language DLL in Delphi, API design and the tools used.

Then, the methods found will be used to implement a prototype of the Delphi SDK. The methods include automatic UML-to-Delphi conversion, translating Java code to Delphi by hand and programming the low-level and high-level interfaces appropriately. Evidently, source code of the Java SDK is studied to implement the Delphi SDK similarly and the ANSI C Stack source code is examined for the low-level interface. In addition, best practices of software engineering are used. Embarcadero RAD Studio IDE is used for the actual programming, the UML diagrams depict software architecture and flow of action, while the memory manager FastMM is used to detect any memory leaks in Delphi.

The implemented Delphi SDK prototype is also tested by developing a simple OPC UA client test application and using it to perform typical OPC UA cases. This way it can be verified that the prototype works, and thus, the conversion methods work.

1.4 Structure of the work

The first chapter presents background knowledge on the research topic and defines the objectives and the scope of the thesis. An overview of OPC UA is given in chapter 2, focusing on relevant topics in context of implementing the client SDK prototype. Chapter 3 examines the development methods needed in the conversion: translating Java code to Delphi, using a DLL written in C, creating a good API and the development tools. Chapter 4 defines requirements for the implemented prototype as a set of use cases. It is followed by chapter 5, where the Java SDK and the ANSI C Stack are studied in order to design the Delphi client SDK prototype to meet the requirements. In chapter 6, these use cases are implemented and tested. Finally, the conclusions are discussed in chapter 7 .

2 OPC UA

2.1 Overview

A comprehensive overview of OPC UA is presented in book *OPC Unified Architecture* [21]. In the early nineties, a common standard for accessing data of automation systems did not exist in automation industry. Instead, different vendors and devices used various protocols and interfaces making development of automation software demanding since drivers had to be tailored for each device to enable the communication. The most successful solution was provided by OPC Foundation in the form of the OPC Classic specification, which presented a commonly usable interface for accessing data of systems and devices. OPC UA is the successor to OPC Classic and was launched in 2006 providing multiple improvements such as a more advanced security model as well as platform and language independence, whereas OPC Classic was built on Microsoft's COM (Component Object Model) and DCOM (Distributed COM) technologies. OPC Foundation verifies the compliance of OPC UA products to ensure they meet the specifications and are, thus, interoperable with each other.

2.1.1 Address Space and accessing Nodes

OPC UA implements a client-server structure, which means that an OPC UA client connects to a server that exposes its information to the client [21]. This information is known as an Address Space, which is made up of Nodes that are identified uniquely by NodeId identifiers and connected to neighbouring Nodes with References. Nodes represent different types, including Objects that are used to structure an Address Space and Variables that contain values. Typically, a client browses Objects of an Address Space to find the wanted Variables whose values are of interest. These values can then be read, written and subscribed to, which means the client will get notified when values change. To read and write values, a client needs to know the NodeId of the Variable Node that contains the value. Subscribing requires also creation of a Subscription to group information sources and a Monitored Item to manage an information source by mapping it to a Node. Then, a piece of information, known as a Notification, is delivered to the client indicating of data value changes in the specified Node.

2.1.2 Services

OPC UA follows SOA (service-oriented architecture), meaning simplified that its functionality consists of independent services [21]. To use a service, a request is handed to a service provider which in turns returns a response message. In this case, as OPC UA implements also a client-server structure, the client creates a request and sends it to the server, which in turn handles the request, creates the corresponding response message and sends it to the client. The service sets are listed in table 1 and the individual services are defined thoroughly in the OPC UA specifications [26].

Table 1: Service sets of OPC UA [26].

Use case	Service sets or services
Find servers	Discovery Services Set
Connection management between clients and servers	Secure Channel Service Set Session Service Set
Find information in the Address Space	View Service Set
Read and write data and metadata	Read and Write Service
Subscribe for data changes and Events	Subscription Service Set Monitored Item Service Set
Calling Methods defined by the server	Call Service
Access history of data and Events	HistoryRead HistoryUpdate Service
Find information in a complex Address Space	Query Service Set
Modify the structure of the server	Node Management Service Set Address Space

2.1.3 Application architecture

OPC Unified Architecture describes an abstract application architecture, where a functional OPC UA software comprises three layers of software: the low level API Stack, the high level API SDK and the application layer on top where a client or a server are used [21]. When the client-server architecture is applied with the layers, the complete OPC UA application architecture looks as is illustrated in figure 2.

As can be seen in the figure, all the transportation and the security-related

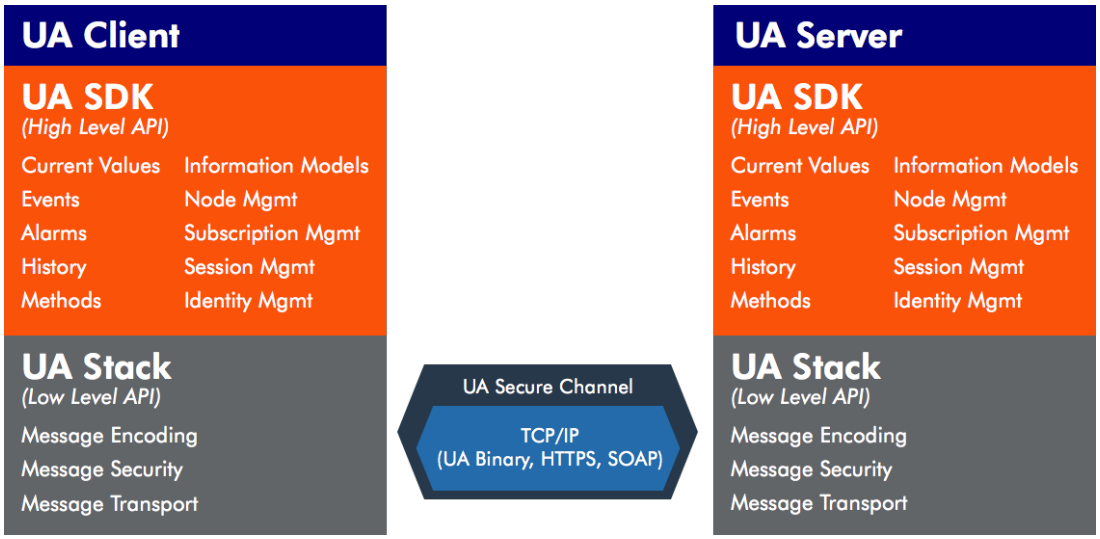

Figure 2: Application architecture of OPC UA [36].

handling of messages between an OPC UA server and a client happens at the Stack level. A Stack provides a basic API for accessing the OPC UA services, while an SDK provides an API more suitable for typical application use cases hiding the Stack layer functionality.

2.1.4 Security

Whilst the OPC UA security model offers various possible configurations to be used in specific circumstances, it is also complex to a certain degree emphasizing the need for both an OPC UA application and especially an SDK developer to have an understanding of how it is modeled. Within this thesis, a basic understanding is needed to comprehend how the client prototype can connect to a server securely.

The security architecture is presented in figure 3. It consists of three layers, each of which is responsible for providing specific security related characteristics that are implemented by various security mechanisms. *OPC UA Specification Part 2: Security Model* defines the security model in detail, including also definitions of the characteristics in the context of OPC UA [24].

At the bottom there is the transport layer representing *availability*: to enable exchange of information between two parties, a Socket Connection is created using either the more common UA TCP (Transmission Control Protocol) or HTTP/SOAP for Web Service environments, which are the two transport protocols supported by OPC UA.

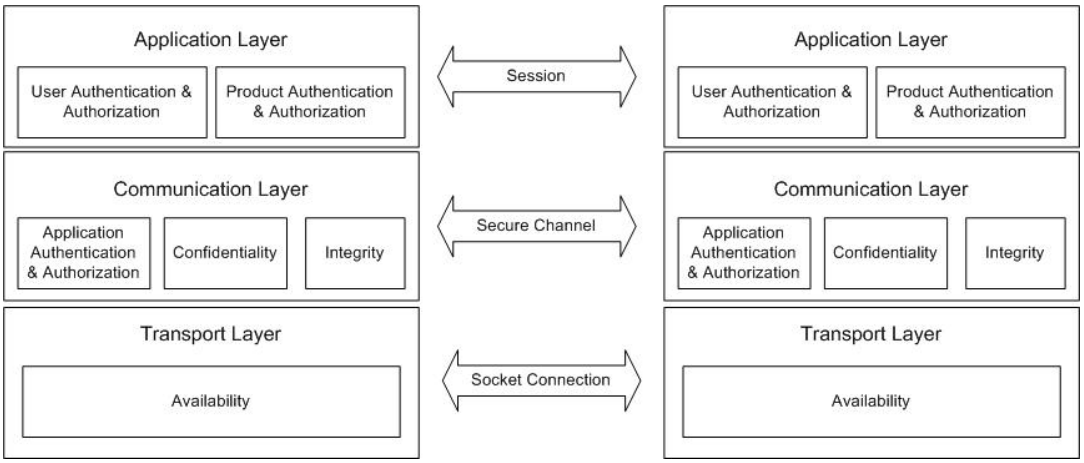

Figure 3: Security architecture of OPC UA. [21]

At the second layer, a Secure Channel is formed on top of a Socket Connection to provide the communication security mechanisms. First, when creating a Secure Channel, both a client and a server application identify themselves by exchanging digital Certificates that are unique for each OPC UA Application Instance. This allows both sides to *authenticate* the other party by checking the received certificate and then to *authorize* the creation of a Secure Channel, if the certificate, and thus the application, is trusted. Furthermore, messages are secured at the Communication Layer according to the chosen OPC UA Security Mode:

- **None** is used, when no securing is wanted.
- **Sign** specifies that messages should be signed with the Private Key of an Application Instance Certificate of an OPC UA client. Signing provides *integrity* by implying that the message was not modified during the transmission.
- **SignAndEncrypt** specifies that, in addition to signing messages as declared above, they should also be encrypted with the Public Key of an Application Instance Certificate of an OPC UA server. This prevents unwanted parties from reading the messages, thus providing *confidentiality*.

The algorithms used for signing and encryption are defined by Security Policy. The third layer is the Application Layer that is responsible for providing a Session, which identifies the user of a client and a product. *Authentication* and *authorization* of a product is very similar to the one of an Application Instance at the communication layer. A Certificate is used for identification, and based on whether it is trusted or

not, the party that received the certificate can decide to continue the session or to close it. For example, the version number of a software can be recorded in a product certificate and used to determine if the software of the client and the software of the server are compatible together. A user can be authenticated with four types of credentials: a certificate, a user name and a password, a so called security token and anonymous authentication.

2.2 Previous implementations

2.2.1 OPC UA libraries

As was mentioned in the introduction, there are no available Delphi SDKs for OPC UA. However, OPC Foundation has certified existing OPC UA SDK implementations for OPC Compliancy in other languages. These include the following:

- OPC UA Embedded Server SDK by MatrikonOPC is aimed to be used in embedded devices such as Programmable Logic Controllers (PLC), sensors, drives and servo-Amplifiers [3]. It is a commercial product written in C language.
- OPC UA .NET Server and Client Development Toolkits from Softing are for commercial use and to be used with C# [4].
- Unified Automation delivers three commercial SDKs: the ANSI C SDK is aimed for embedded and most portable needs, the C++ SDK is for software where cross-platform support or high performance is needed and the .NET SDK is for MS environments [5].
- Prosys Java SDK is a commercial product for platforms running Java SE 6-8 and includes a code generator that creates Java code based on existing OPC UA models [39]. This is the SDK that is converted into the Delphi SDK and will be studied further in chapter 5.

OPC Foundation delivers three stack implementations to verify the low-level interoperability between OPC UA applications. They are for ANSI C, .NET and Java environments, all of which are provided as open-source software.

2.2.2 Prosys OPC SDK Sentrol

Prosys PMS Ltd has previously developed a client and server SDK for developing OPC Classic applications, commonly referred to as Prosys Sentrol. It is written in Delphi and can be used with Embarcadero's Rapid Application Development (RAD) Studio, both with the Delphi and C++ Builder of the Studio [38].

Many Prosys Sentrol users would benefit from upgrading from OPC to OPC UA. To make this transition as effective as possible, an interface is planned on top of the OPC UA Delphi SDK to provide application developers an API similar to the one of Prosys Sentrol. The interface will be designed and implemented in the further development phases after implementing functionality similar to the Java SDK. Thus, exploring how the interface would be implemented is out of the scope for this thesis, but a brief introduction is given since the RAD layer may affect API design choices later.

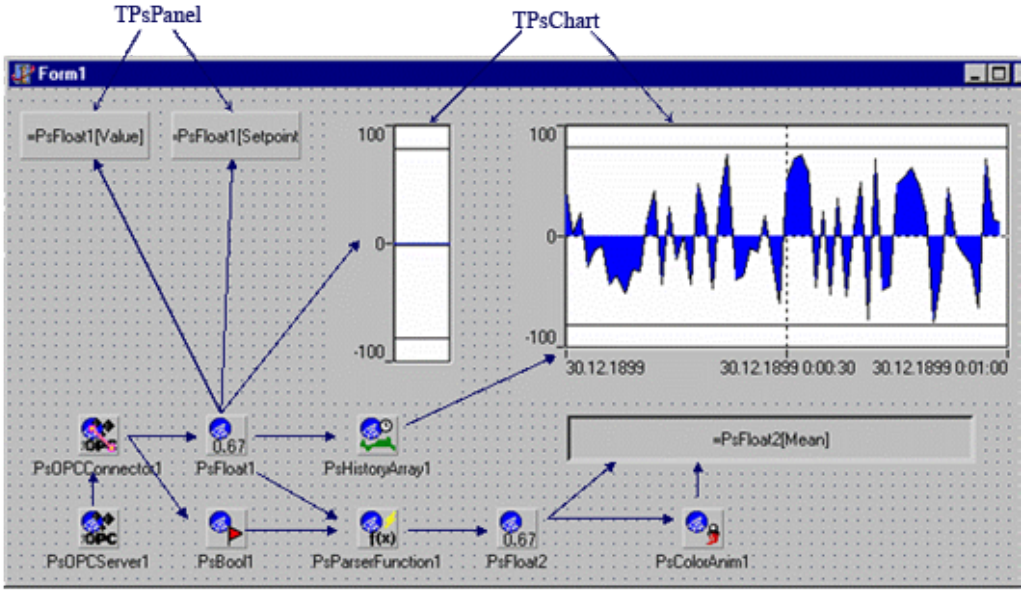

Figure 4: Developing a sample Prosys Sentrol client application. [38]

Figure 5: Prosys Sentrol data access model. [38]

Figures 4 and 5 present a brief introduction to the RAD client model of Prosys

Sentrol. In the picture above, a client application is being developed showing a user interface and the Sentrol components that are recognized with "Ps" and "TPs" as the first letters. The base component, named *PsOPCServer1*, offers the configuration of the server while the connector component *PsOPCConnector1* enables connection and data access to the server. The wanted data in the application is specified with the Sentrol variable components, in the sample named *PsFloat1* and *PsBool1*. Instead of just reading the current value, *PsHistoryArray1* provides access to the history data of *PsFloat1*. *TPsChart* component provides a chart for the application user showing the specific history data. Similarly, *TPsPanel* components are used to show and set the current value of *PsFloat1*.

There are several more components available to aid developing client and server OPC applications with Prosys Sentrol, and the intention is that the corresponding components would be implemented into the OPC UA Delphi SDK. Furthermore, there are also notable architectural differences between the OPC SDK Sentrol and the OPC UA Java SDK implementations, but the impact this has on the development of the RAD interface for the Delphi SDK has not been examined.

3 Development methods

3.1 Translating Java to Delphi

Ideally, the existing Java SDK could be reused effectively by converting it automatically to Delphi with a so called source-to-source compiler. However, since there are no existing Java-to-Delphi compilers to be found, one would have to be implemented by hand. This seems difficult, as translating only a small piece of code to Java from Turbo Pascal, a language closely related to Delphi, fails easily resulting in incorrect program behaviour [2]. The semantics of the languages are too different for performing the translation automatically. The difficulty of developing such complete source-to-source compilers is also indicated by the current state of source-to-source translation in software engineering [1]. Although language translation is a widely used practice nowadays, it is noted that fully automatic and guaranteed correct translations do not generally work. Instead, translating subsets of languages or programs can be profitable in many cases. The major advantages of language translations are considered to be that there is no need to write everything from scratch and that code based on existing reliable code tends to work more reliably too.

Clearly, translating the Java SDK automatically to Delphi is out of the question, but a translation made by hand can be expected to have similar types of benefits. For doing the manual Java-to-Delphi transition effectively, the languages are examined to find out what can be translated directly and which situations require more case-specific consideration. Both are modern object-oriented languages that provide a wide framework to work with, therefore the translation seems to be possible to perform efficiently.

3.1.1 Program and unit structure

In Java, a program comprises multiple .java files that contain exactly one class, not counting its possible nested classes [20]. A Delphi program consists of .pas unit files that can contain as many classes as needed [19]. In practice, classes that provide similar type of functionality should be placed in a single Delphi unit.

In Java, class methods are implemented and variables declared at once inside a class declaration [20]. In Delphi, each unit has a separate declaration area, specified by term *interface*, where the classes and their members are declared, whereas *implementation* specifies the start of an area where methods are implemented [19].

3.1.2 Cross-referencing

Cross-referencing between classes is not restricted in Java. Delphi implements a more hierarchical class structure, meaning that for example circular references between units or classes are not allowed in the *interface* area, however, they are allowed in the *implementation* area [19]. Cross-reference situations can be solved so that one of the classes references to a parent of the other class instead of referencing directly to the class. The parent can be either a class or an implemented interface, the choice should be made specifically depending on the situation.

3.1.3 Class member visibility specifiers

The languages share three similar class member visibility options: *private*, *public* and *protected*. These are the most common ones and typically can be translated to Delphi directly, however, there are differences worth presenting. *Private* members in Java are only accessible by a class itself [20], whereas in Delphi they are also available to other classes within a unit [19]. In both languages, a *protected* member can be accessed from a subclass, but in Java also from the same package [20], while in Delphi also from the same unit [19]. In both languages, a *public* member can be accessed whenever its class can be accessed [20] [19]. In addition, a Java member can be declared without a visibility specifier, meaning that a variable is then accessible to the class itself and to classes within the same package [20]. In these cases the Delphi *private* specifier is clearly the closest since its visible to classes in the same unit, but consideration should be used case by case.

3.1.4 Writing and reading member variables

A significant difference is that in Java class variables are accessed typically by calling getter and setter methods [20]. For example, a variable named *test* could be read with method *getTest* and written with method *setTest*. Variables that can be accessed this way are commonly referred to as properties in Java.

Delphi implements actual properties to classes, meaning that a property is specified to be of a certain type, and is read and wrote with specified methods, which are typically also named beginning with "Get" and "Set" [19]. To write or read a property it is accessed as if it was a variable, which links the operation to the corresponding get or set method instead of directly accessing a variable. Thus, variables accessed with setters and getters in Java should be implemented as properties in Delphi.

3.1.5 Methods

As to methods, Delphi makes a difference in naming convention between ones that return and ones that do not return a value. Methods that return a value are called *functions* and the ones that do not are called *procedures* [19]. Also, being a Pascal-based language, variables used within a method need to be declared locally within a *var* section and exactly before implementation of a function [19]. Overloading is supported in both Java and Delphi [20] [19].

3.1.6 Static and class declarations

Java methods and variables can be declared *static* to imply that they can be accessed without creating an instance of their class [20]. This way, a *static* variable is created only once within a class and not for each object separately, thus all the class objects share the same *static* variable. Delphi provides similar functionality, but instead the *class* keyword is used to describe that a member belongs to its class [19].

3.1.7 Memory management

Java uses a garbage collector practice, meaning simplified that the garbage collector of the Java virtual machine (JVM) manages all objects created within a Java program with *new* command and releases automatically the ones that are not referenced anymore [20].

In Delphi, there is no similar helper utility but memory needs to be allocated and freed manually in two ways [19]. Objects are created manually with *Create* method and need to be freed using the *Free* method. *GetMem* is used to allocate memory of a user-defined size as a number of bytes. This memory needs to be freed with *FreeMem* method. These manual memory management features place requirements also on implementation of object destructor method *Destroy*, which is called during the execution of method *Free*. All the memory allocated in an object needs to be freed when the object is destroyed. Therefore, in the method *Destroy*, all objects within a class need to be freed with *Free*, and if there is a pointer to memory allocated with *GetMem*, this memory should be freed correspondingly with *FreeMem*.

Memory management is an especially critical issue when converting Java to Delphi. Since manual memory freeing practice is not implemented in Java code, each needed freeing action needs to be added for the Delphi version to avoid memory leaks.

3.1.8 Syntax

Going through all the syntax rules and grammar of Delphi and Java is not within the scope of this thesis. As an example, the most common operators and their syntax rules in Java and Delphi are presented in table 2. The increment and decrement operators are missing in Delphi, but the *Dec* and *Inc* procedures provide equivalent functionality [19]. There are evident differences between the syntax rules, creating need to translate the Java notations to Delphi, but all of these can be converted directly since the operators perform identical actions.

Table 2: Operator syntax rules in Java and Delphi [19] [20].

Java	Delphi	Description
=	:=	Assignment
+	+	Addition, string concatenation, unary plus
-	-	Subtraction, unary minus
++		Increment by 1
–		Decrement by 1
*	*	Multiplication
/	/	Real division
/	div	Integer division
%	mod	Remainder
==	=	Equality
!=	<>	Inequality
<	<	Less than
>	>	Greater than
<=	<=	Less than or equal to
>=	>=	Greater than or equal to
!	not	Negation
&&	and	Conjunction
||	or	Disjunction

What can be noted is that Delphi uses words more commonly to describe operators than Java. This holds true in other areas of grammar too, for example, instead of brackets the keywords *begin* and *end* are used to encapsulate blocks [19].

3.1.9 Data types

The primitive Java data types and their equivalent Delphi types are listed in table 3 to present the types that can be translated directly.

Table 3: Corresponding data types between Java and Delphi [30] [20].

Java	Delphi	Description
boolean	Boolean	True or false
char	Char	16-bit character
byte	ShortInt	8-bit signed integer
short	SmallInt	16-bit signed integer
int	FixedInt	32-bit signed integer
double	Single	32-bit floating point
long	Int64	64-bit signed integer
float	Float	64-bit floating point
string	String	Unicode string

3.1.10 Java listeners and Delphi events

The listener pattern is used in Java for reacting to specific events with customizable means [20]. A class where the events of interest occur contains a number of listener interfaces whose methods are called when the events take place. So, to react to the events, a listener interface first needs to implemented as a class. Then, an instance of this class is attached to the object whose events are of interest. In turn, a Delphi event is a method pointer that is linked to a method where the handling of an event is wanted to occur [29]. Typically they are implemented as class properties without getter and setter methods.

Even though the listener pattern can be applied in Delphi, the event pattern is a more appropriate practice in Delphi since there is no need to create additional objects that would later need to be freed, making memory management easier. Thus, the listener practice in the Java SDK should be converted to the event pattern for the Delphi SDK.

3.2 Wrapping a C DLL in Delphi

3.2.1 Calling C functions

Dr. Bob's Delphi Clinic provides instructions on calling C DLL functions in Delphi [40]. Typically, first a separate wrapper unit is created to use a single DLL as if it was a normal Delphi unit. Then there is the choice of loading the DLL either statically, meaning the DLL is loaded at once when the program starts, or dynamically, meaning the DLL is loaded only when there is the need to use its functions and can be unloaded after that. The static loading is presented here, because it is more relevant in the case of Delphi SDK where the most important functionality is dependent on the DLL of the ANSI C Stack. Each DLL function to be called needs two definitions in the wrapper unit following the Delphi interface-implementation pattern presented in section 3.1.1. The definitions are illustrated with the following sample:

```
Interface
  Function Subtract(var Original, Subtract: Double): Integer; CDecl;

Implementation
  Function Subtract; External 'Math.DLL' Name '_SubtractTest@4';
```

In the implementation part, directive *External* followed by a file name is used to tell the compiler that the called Delphi procedure or function is implemented in an external object file or DLL. The file name can be followed by *Name* directive and the actual function name in a C DLL to wrap the C method in Delphi with a different name [19]. In the interface, methods are defined with their parameter names and data types, where especial attention is needed to make sure that the Delphi data types match with their C counterparts (see 3.2.3). The last directive is *CDecl*, which tells the compiler that a function should be called using C-style calling conventions. The above example would be suitable for calling a C DLL function defined in a header the following way:

```
__declspec(dllexport) int _SubtractTest@4(double * original, double subtract);
```

3.2.2 Memory management

An important issue of working with a C DLL in Delphi comes with memory management. Both languages use their own run-time libraries (RTLs) [28], which results in a Delphi executable and a C DLL having their memories managed separately [32]. This means that memory allocated in one module must also be freed in the same module. If an executable tries to free memory that was allocated in a DLL or vice versa, problems such as heap corruption and access violation errors start to occur. However, an executable can free memory allocated by a DLL when wrapper functions for freeing memory are provided in the DLL [33].

3.2.3 Type translations

Usually, definitions of C DLL functions and used data types can be found in C header files, after which the functions and the data types can be translated into their respective Delphi forms. Essential conversions include the following [27] [19]:

- **C functions** translate to Delphi procedures and functions in a similar way methods in Java do. A C function without a return value becomes a procedure, while one with a return value becomes a Delphi function.
- **C function pointers** can be converted to standard Delphi method pointers.
- **C pointer parameters** can be passed as a reference to a variable with two keywords. The *var* keyword implies the value of the variable will be changed in the method, whereas *const* references can only be used for reading the value. In addition, Delphi pointer values can be passed as parameters.
- **C structs** are records in Delphi.
- **C enumerations** can be translated directly to Delphi enumerations.
- **C const** types can be defined as standard Delphi const types.
- **C integer type error codes** are typically used to specify errors in C, since the exception practice is not implemented in C. These should be converted into Delphi exceptions and raised.

In addition, the primitive data types need to match exactly between the C and Delphi code definitions. If the size of a data type is only one byte wrong or in an incorrect Delphi format, strange and difficult-to-trace errors may occur. The equivalent data types are listed in the following table:

Table 4: Corresponding data types between Delphi and C [30] [27] [7].

C type	Delphi type	Description
unsigned char	Byte	8-bit type, guaranteed to be positive
char	ANSIChar	8-bit type
unsigned short unsigned int	Word	16-bit unsigned integer
short int	SmallInt	16-bit integer
void*	Pointer	32-bit pointer
unsigned long	LongWord	32-bit unsigned integer
long	FixedInt	32-bit integer
float	Single	32-bit floating point
unsigned ___int64	UInt64	64-bit unsigned integer
___int64	Int64	64-bit integer
double	Double	64-bit floating point

It is worth noting that the C data types short, int and long, whether unsigned or not, contain at least the defined 16 or 32 bits of data depending on the compiler [27]. Thus, they can also be bigger than their size in the table. However, the ANSI C Stack is compiled into a DLL by hand, therefore the sizes can be confirmed in the compilation phase.

3.3 Creating a good API

An API exposes functionality of a software library, such as an SDK, to application developers. The importance of creating a good API can be understood by observing the effects of APIs that are hard to understand and use. Bad APIs result in increased application development time and cost as well as crashing applications for end users [17]. Again, this leads to an increased need for support and maintenance, whereas a good API is not that demanding for the API developer company itself but attracts more customers and encourages established customers to invest more in the product [13].

An important aspect of API design is to put effort on creating a good API at once. The motivation is that when the code has been used in application development, making changes to the API may also require updates for all the applications using the code. A poor interface would need to be updated continuously, making development of applications using the interface inefficient. From an economic point of view, a bad interface could seriously cut down the lifespan of a product [8].

Therefore, in the context of the Delphi SDK, special effort should be placed on converting the Java SDK API to Delphi. Even though the objective is to build the Delphi SDK by converting the successful Java SDK very directly to Delphi, putting focus on how a good API can be implemented should help to ensure that the quality achieved in the Java implementation is maintained in the Delphi version and to identify where improvements of some type could be made, since making changes later is more difficult. Moreover, differences between the languages also have an impact on the API and good design practices are expected to help solve these situations.

There are books about API design in specific programming languages, not Delphi, but such as C++ [8] and Java [16], which are examined as they also address designing APIs in general and the languages concerned share common characteristics with Delphi. In the past decades, APIs also have been a subject for several studies highlighting their significance in software engineering.

3.3.1 How to make a good API

It is commonly remarked that designing an API is similar to designing a graphical user interface (GUI) in the sense that an API is a user interface for programmers [17] [9]. The key characteristics of a good GUI, user-centered design and good usability, are also the bottom lines in creating a good API. The favored approach is to build use cases (scenarios) and to identify the requirements for performing a specific use case from the user perspective [8] [9] [13] [16] [17]. These requirements are then used in design, instead of letting the implementation guide.

Among other qualities, an API with good usability is easy to learn and use, hard to misuse, consistent in its practices and easy to extend [13] [16]. These characteristics, as well as 12 cognitive dimensions presented by Clarke [9], can be used to evaluate how usable an API is. Typical problems of APIs include incomprehensible high-level design and unsuccessful documentation, which can be solved by providing especially code examples along with snippets, tutorials and sample applications [18]. However, documentation needs to be done carefully as incomplete or imprecise documentation is a common source of problems [11].

Some principles and practices are proposed for making a good API. These can be grouped into practical and general guidelines, of which the practical ones are more applicable regarding the translation of the Java SDK to Delphi. They are presented in the next sections, while the general guidelines consist of the following among others:

- Backward compatibility: an application using an API should continue working with as small changes as possible when the API is updated to a newer version [16].
- An API should be only as big as what is needed for its functionality [13].
- Importance of naming: abbreviations should be avoided and names should be self-explanatory [13].
- Implementation should be separated from the API [8].
- "Document every class, interface, method, constructor, parameter, and exception" [13].
- Accessibility to information should be minimal [13].

3.3.2 Required constructors

Jeffrey Stylos and Steven Clarke performed a study on requiring parameters when constructing an object [10]. The studied constructor patterns are depicted in figure 6. The conclusion was clear: programmers preferred a *create-set-call* pattern over *required constructors*. When the only available class constructor required parameters, most programmers tried initializing the class with null or empty objects as parameters resulting in various types of errors. Importantly, it was also noted that having an optional constructor with parameters did not have a negative effect, instead some programmers preferred a parameterized constructor when it was given as an option.

Based on these observations, classes should always provide a default constructor. Consequently, the Java SDK classes without a default constructor should be implemented with one in the Delphi version if possible concerning the construction process.

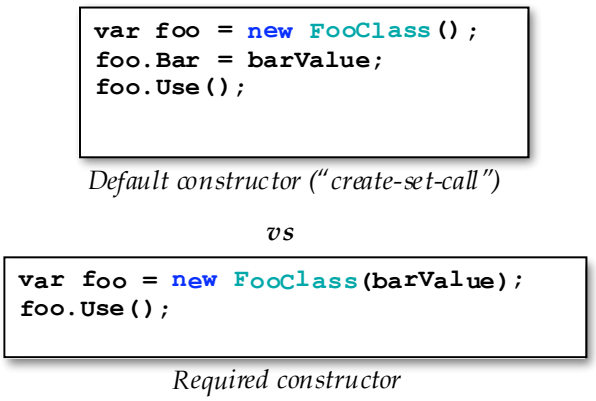

Figure 6: Compared object construction patterns. [10]

3.3.3 Constructors and factory methods

In addition to creating a class instance using a default constructor, also more advanced approaches are used having their specific benefits. Especially factory methods are commonly used, either in the instantiated class itself statically or in a separate class, typically also as static methods. The advantages of factory methods include being able to name the methods unlike with constructors, not needing to create a new object if not necessary, the ability to create a subclass instead of an exact class as well as better possibilities to synchronize the object creation process [12] [14]. The main benefit of factory methods is offering more flexibility to the implementation

of construction. Flexibility can be especially useful when the construction process is complicated, the class requirements are expected to change in the future or the implementation details are wanted to be hidden from developers.

However, two API usability studies on the factory pattern had very similar results indicating that constructors should be preferred over the factory pattern [12] [15]. According to both studies, developers primarily seek to create an object through constructors, while finding a factory method takes more time and creates confusion. After learning that factory methods are used, constructor and factory approaches were preferred more equally. A proposed good solution would be to have both options available to fit different programming styles [15], but as is noted, having two explicit choices creates problems in trying to identify the correct choice [12].

In conclusion, constructors should be preferred. Factory methods should be considered when an instantiation process is more complex or affected by external factors. In addition, special attention should be given to documentation to point out when the factory constructor practice is applied [14].

3.3.4 Methods and parameter count

The consensus is clearly to avoid long parameter lists: writing a method call with many parameters evidently takes longer than with no or a few parameters [15] [13] [8]. The preferred amount is up to 3-4 parameters, while with many parameters, the suggestion is to create a container class for some of the parameters. The problems arise also from optional parameters that are irrelevant to the application developer, leading into hesitation about finding the proper values and many times into searching for a more suitable method overload. Overloading methods with different parameter combinations was found to be a solid solution for users [15], even though having multiple choices to choose from might seem impractical. However, with more than a few optional parameters the number of parameter combinations, and therefore the number of required overloads, becomes hard to manage and a builder pattern should be considered [14].

3.3.5 Amount of class members

Perhaps unexpectedly, having 150 class members instead of 50 results only in a small performance impact when users search for class methods [15]. The reason is that developers tend to search for methods by writing the expected prefix of

a method name and an integrated development environment (IDE) proposes the suitable options. However, as is noted in the study, this highlights the importance of naming since the class member list needs to be browsed through if the wanted method is not found with the prefix.

As was remarked previously, an API should not be bigger than what is needed. But clearly, there is no reason to overly avoid adding class members when they are considered useful.

3.3.6 Minimizing accessibility

The general principle of minimizing accessibility is typically applied to classes and their members [8]. Primarily, this means defining every class variable and method private when possible. The reason is clear: whenever users are given access to information, they tend to start using this information whether it is meant to be used or not. Then, modifying these parts of the API becomes more difficult since users and their applications are dependent on it. The same principle applies also to internally used classes, which should be kept within the implementation.

Secondly, there is the matter of avoiding direct access to public members, instead getter and setter methods should be used. Reasons include the possibilities to set variables as read-only or write-only, validate set values, notify another module of a value change and control whether a return value should be a copy of the original or a reference to it. This is the same in languages such as C# and Delphi, where class properties should be used to define the get and set actions, which are typically the getter and setter methods.

These observations suggest that class member visibilities should be considered private according to whether a member is needed for the API or not and that public variables should be accessed with getters and setters or with properties. It is expected that these principles are well applied in the Java SDK, but it is better to make the transition to Delphi case by case. Also, as was already suggested in section 3.1, Java setters and getters should be converted to Delphi properties.

3.4 Development tools

3.4.1 UML

The Unified Modeling Language (UML) is a modeling language designed for software development purposes [35]. It defines several diagrams for specific uses cases, and on the top level the diagrams are grouped into two categories. *Structural* diagrams illustrate how things, such as classes or objects, are related to each other in a software and are used to present software architecture. *Behavioural* diagrams portray a flow of actions and are designed to represent how a use case is performed or how a process is executed. In this thesis, the UML diagrams are used to represent both architecture and flow of actions.

3.4.2 Embarcadero RAD Studio

Embarcadero RAD Studio is an IDE for developing Delphi and C++ applications [31]. It provides a native cross-platform framework and enables development of applications on Windows 10, Android, iOS and OSX platforms using same UI with FireMonkey. It also includes a visual component library (VCL) for rapid Windows application development purposes. RAD Studio is used to develop the OPC UA Delphi SDK.

3.4.3 FastMM

FastMM is a custom memory manager that provides comprehensive memory leak reporting in both Delphi and C++ environments of RAD Studio [41]. It allows configuration of the memory error types that are detected and writes detailed reports in a file. It is used for debugging and testing in the development of the Delphi SDK, replacing the default Delphi memory manager on these occasions.

3.4.4 OpenSSL

OpenSSL is an open-source library that provides security functionality for communication protocols as well as general-purpose cryptographic functions [34]. It is used for the Delphi SDK to provide required functionality for handling Certificates. Since it is written in C and can be found as a DLL, the methods presented in chapter 3.2 are applicable.

3.4.5 ModelMaker

ModelMaker Pascal Edition by ModelMaker Tools is used in the thesis for drawing UML models and to generate the initial Delphi Client SDK class architecture that is based on the Java SDK. The purpose of the software is summarized on the web page of the company [6], "ModelMaker is a two-way class tree oriented productivity, refactoring and UML-style CASE tool." In practice, this means the program is integrated with Delphi and instead of just generating Delphi code, also the modifications made in Delphi are updated to ModelMaker's UML diagrams.

3.4.6 Prosys OPC UA Simulation Server

Simulation Server is a stand-alone OPC UA Server that supports, among other features, security and data configuration and provides comprehensive logging information [37]. These features make it a suitable tool also to test the Delphi SDK with. Especially the easy way to see connection logs is considered useful for solving debugging situations. It also provides simulation data as static and consequently changing values that can be used to test the data access Services.

4 Requirements

The requirements for the implemented client prototype are defined in three parts. First, a set of use cases the client needs to be able to perform is selected to specify the required scope of the implementation. This is followed by going through each of the use cases and specifying how an application should be able to perform them. Thus, by defining the use case requirements, the user-centric approach presented in chapter 3.3.1 can be applied to initiate the client prototype design. Finally, requirements for the low-level interface with the Stack are defined.

4.1 Required use cases

The prototype needs to be developed sufficiently to provide enough information about developing the OPC UA Delphi SDK, but defining the exact scope is very abstract. After all, the only way to guarantee that there would emerge no unexpected issues would be implementing the full SDK. However, minimum requirements are defined as a set of specific use cases the prototype needs to be able to perform.

Since the purpose is to build the Delphi SDK upon the class architecture of the Java SDK and to find out the issues created by such a conversion, the most essential classes of the Java client SDK should be implemented in Delphi enough for them to be used in some circumstance. Furthermore, interaction with the ANSI C Stack needs to be examined. Multiple Stack methods, especially Service methods, should be called and callbacks should be received from the Stack. As use cases are used as requirements, there will evidently be material to evaluate creation of the API.

By studying the Java SDK, implementing the following four of the most common OPC UA use cases fulfills these needs, as is seen in the next chapters that focus on the use cases:

1. Connecting securely to a server.
2. Browsing an Address Space.
3. Reading and writing Node values.
4. Subscribing to a Node.

4.2 Performing the use cases

Since the Delphi SDK is built on the architecture of the Java SDK, it is also supposed to be used in a similar way. This way developers familiar with the Java implementation are able to start developing OPC UA applications in Delphi very effectively. This requires the API between the Delphi SDK and a Delphi OPC UA application to be similar to the one of the Java version. However, making an exact translation is not the goal as the differences between the two languages and the intention to make improvements compared to the Java SDK also affect how the ideal Delphi SDK API should be implemented. It can be defined that performing the use cases in Delphi needs to possible by setting the equivalent parameters and calling the corresponding methods than in the Java version. Performing these use cases in the Java SDK is illustrated with the UML sequence diagrams, which are suitable for depicting interaction between different classes.

4.2.1 Use case 1: Connecting securely to a server

Connecting to a server is the first step before any of the other use cases can be performed. To connect to a server several parameters need to be set, especially when the connection is wanted to be secure. These initializations are also examined here and required from the implementation.

First, an instance of class *UaClient* is created and initialized with a server URL. As can be seen in figure 7, *UaClient* is the main class for creating a client application providing all the client functionality. It is followed by setting the Security Mode of the client to *BASIC256SHA256_SIGN_ENCRYPT*, which implies the messages will be signed and encrypted using a 256 bit version of Secure Hash Algorithm (SHA).

Then, an *ApplicationDescription* object is created and set with four parameters that specify an application [26]. This is followed by creating an instance of class *ApplicationIdentity* that combines the *ApplicationDescription* object and additional security-related settings of the application. Static factory method *loadOrCreateCertificate* is used for the creation taking several parameters: the *ApplicationDescription* instance, an organization name, a Private Key password, a Private Key file path, KeyPair of Issuer Certificate to provide both a Private Key and a Certificate if available, and finally a boolean value to tell if a certificate is loaded and should be renewed in case its validation time has passed. After this, the *ApplicationIdentity* instance is set to identify the *UaClient* object.

Figure 7: Sequence diagram of the connect use case.

A *UserIdentity* instance is then created to provide a user that can be authenticated with a name and a password. It is set for the client that is finally initialized enough to establish a secure connection by calling method *connect*.

4.2.2 Use case 2: Browsing an Address Space

Finding information on a server is performed by browsing its Address Space. In this use case, it is required that the root Node can be identified and used to start browsing the Address Space from the starting point.

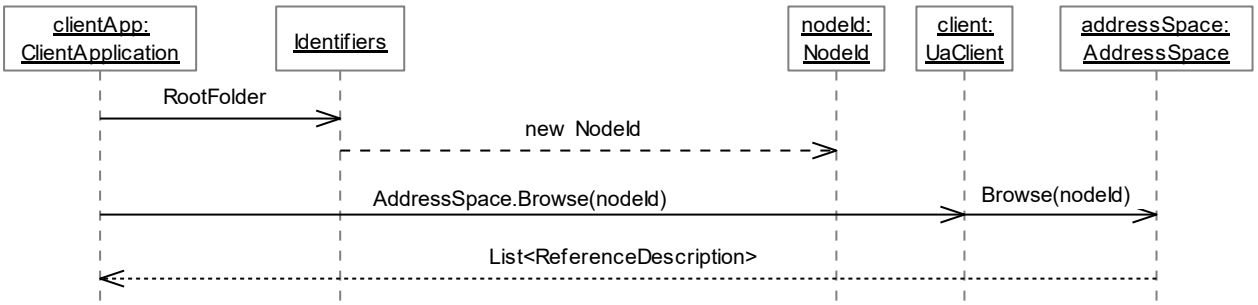

Figure 8: Sequence diagram of the browse use case.

Figure 8 depicts the necessary steps for browsing a root Node. First, class

Identifiers is specified with the root Node identifier *RootFolder*, which results in a static construction of the wanted *NodeId* instance.

The *Browse* function is accessed through the other main class of the Java SDK, *AddressSpace*, which has an instance in the *UaClient* class as a field. The *NodeId* instance pointing to the root Node is given as a parameter and a list of *ReferenceDescription* instances should be returned pointing to the Nodes that are connected to the root Node.

4.2.3 Use case 3: Reading and writing Node values

Reading and writing Node values is perhaps the most typical use case in context of OPC UA. Here, it is required that a specified Node can be written and read using Integer values. Figure 9 presents the Java API of the use case.

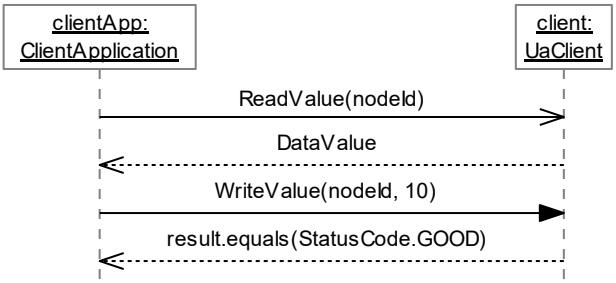

Figure 9: Sequence diagram of the read and write use cases.

To read a Node value, method *ReadValue* is called and the Node is specified with a *NodeId* object. Then, an instance of the OPC UA defined type *DataValue* is returned, providing additional information to the actual value [26]. Writing a Node value is performed with the *WriteValue* method. The wanted Node is identified first, followed by the value wanted to assign to the Node. The returned value is true if the write operation was executed already and false in any other case.

4.2.4 Use case 4: Subscribing to a Node

In a typical scenario, the Node data values, and especially the changes in them, are of interest to an application. Subscribing to Nodes allows applications to be notified of value changes, instead of having to repeatedly read the values of the wanted Nodes. It is required that a specified Node can be linked via corresponding *MonitoredDataItem* and *Subscription* objects to a customizable listener, thus allowing reacting to the changes in the Node value.

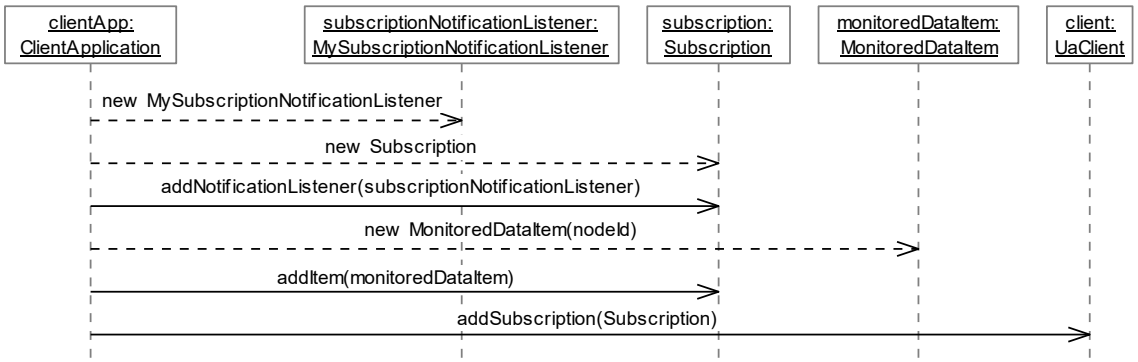

Figure 10: Sequence diagram of the subscribe use case.

Figure 10 illustrates the steps for Subscribing to a Node in the Java SDK. First, to react to the changes of data values, a *SubscriptionNotificationListener* interface (not depicted) needs to be implemented in an application side class. In this use case, the interfaced class is named as *MySubscriptionNotificationListener*, and creating an instance of it is the first step. Next, an instance of the *Subscription* class is created and the *MySubscriptionNotificationListener* object is attached to it to listen to the Subscription Notifications. Then, a *MonitoredDataItem* object is created with a *NodeId* to specify the Node to subscribe to. It is then attached to the Subscription that in turn is attached to the *UaClient* object. As a result, the client will start monitoring changes in the Node and notify the *MySubscriptionNotificationListener* instance about them.

4.3 Requirements for the ANSI C Stack interface

Ideally, when using an external software library, the API provided by the library would be enough for using the library effectively, including possibly header files or similar containers that define the functions and the data types in the application development language. This is not the case with the Delphi SDK using the OPC UA ANSI C Stack. Not only are the languages different, but the ANSI C Stack also provides data in an unusable binary format. As a result, these issues affect the development in ways that can be grouped into inevitable and required. Inevitably, the needed functions and data types of the ANSI C Stack need to be defined in Delphi to be able to use them, which was discussed in section 3.2. What is required is conversion of types that are not easily usable in Delphi into types that are. Most of the basic data types do not need to be converted between the Delphi SDK and

the Stack, but there are two specific cases where a conversion is needed:

OpcUa_String is a UA binary string format and cannot be handled as a standard Delphi string. It should first be converted from the binary format to a raw PAnsiChar using the Stack function OpcUa_String_GetRawString and then to a standard Delphi string which can be performed with an implicit casting.

OpcUa_StatusCode is an integer type that specifies an OPC UA error code. As discussed in section 3.2, it should be converted into a corresponding exception type and raised.

In many cases, a variable of either of these types is part of a C structure and respectively later of a Delphi record. In these cases, it is required that the whole record is converted into a class or a record that contains only standard Delphi types.

Furthermore, there is the question of where these conversions should happen. It is especially crucial to provide an application developer the standard Delphi data types and practices, after all, the purpose of an SDK is to make developing an application easier. But as the Delphi SDK is a Delphi software itself, its development is also more straightforward if the data from the Stack is converted so that it can be handled using standard Delphi methods. In addition, changing the ANSI C Stack to another Stack library should be made as effortless as possibly. A possible further development project is creation of a Delphi Stack, which would have the benefits of not being dependent on a third-party library and a remarkably reduced amount of needed data conversions in the SDK, resulting in code that is more readable and easier to maintain. Changing to a new library is the more difficult the more there are dependencies to the old one, hence these dependencies should be kept to a minimum.

These remarks suggest that, whenever possible, the ANSI C Stack types should only be used when interacting directly with the Stack, resulting in the following conclusions:

- ANSI C Stack specific data types should only be used when calling a Stack function or receiving a callback from the Stack.
- Conversions between the Delphi SDK and the Stack types should be performed instantly before or after a function call or callback.

5 Design

The design of the Delphi SDK follows the design of the Java SDK, but differences between the two languages and the Stacks used have an effect on the final design. As in chapter 4, the UML sequence diagrams are used for illustrating use case APIs. Then, the UML activity diagram was chosen to depict complex processes in the SDK layer since it is very suitable for demonstrating a flow of actions without being affected by the classes the actions happen in, while the sequence diagram becomes very obscure when functions are called mostly within one class. Class structures are illustred with the UML class diagrams.

Each single action in an activity diagram represents an actual method called during the process, but since the complete connection process is complex and wide, not all method calls nor steps are modeled. The intention is to give an overview of the most relevant steps and highlight the differences between the Delphi and the Java SDK implementations in terms of what is needed to add and what can be dropped in the conversion. The following notations in the UML diagrams are used to guide the implementation:

- **Violet color** indicates a method or a class is a part of the SDK layer.
- **Green color** implies that a method or a class belongs to the Stack layer.
- (+) guides that a method exists in the Java Stack but not in the ANSI C Stack, therefore a similar method should be implemented in the Delphi SDK.
- (?) means that a similar method exists in the ANSI C Stack but is not used in the samples, proposing that the method should be tested and then probably implemented by hand for the Delphi SDK.
- (-) marks a method that is implemented in the Java SDK but should be dropped in the Delphi SDK.

5.1 Client architecture

The Delphi SDK client architecture is based upon the client architecture of Prosys Java SDK. The intention is that the Delphi SDK is easier to develop with for developers familiar with the Java SDK and more practical to maintain at Prosys since SDK developers only need to be familiar with one design. Figure 11 presents the design of the Delphi client SDK architecture. The most notable classes for an application developer are *TPsUaClient* and *TPsAddressSpace* that provide most of the client functionality. The other classes can be accessed by a user for utility purposes.

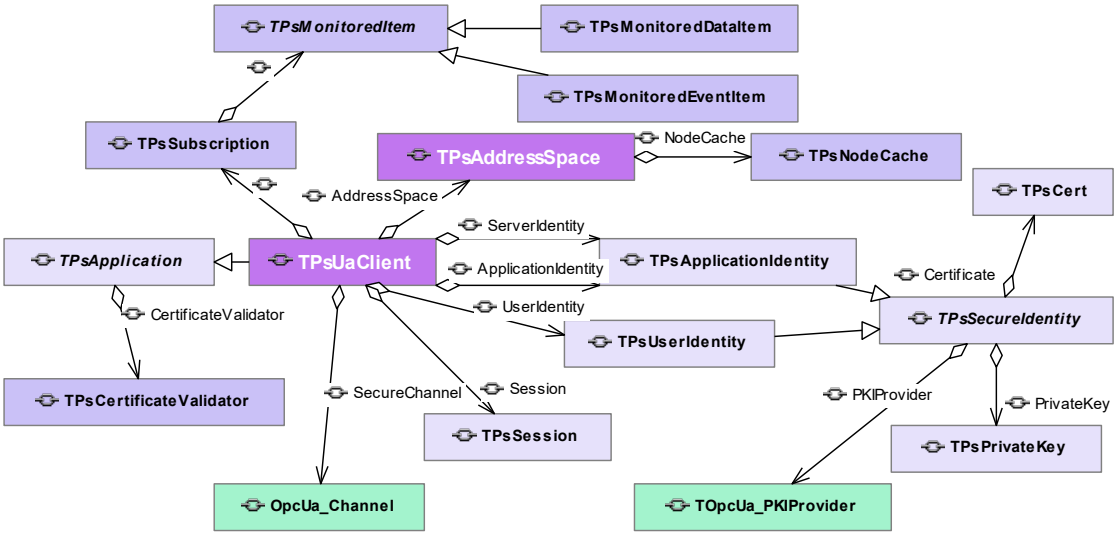

Figure 11: Client architecture of the Delphi SDK.

There are only a few modifications in comparison to the client architecture of the Java SDK, which is why it is not presented in this thesis. The classes representing Session, Certificate and Private Key need to be implemented by hand in the Delphi version since they are not available in the ANSI C Stack, thus they are presented on a violet background color instead of green. Also, *OpcUa_Channel* is a pointer handle to a Secure Channel provided by the ANSI C Stack instead of an actual class. In addition, a *TOpcUa_PKIProvider* ANSI C Stack type is used for loading and saving both Certificates and Private Keys. With the Java Stack, these actions could be performed using its corresponding *Cert* and *PrivateKey* classes.

5.2 Converting ANSI C Stack types to Delphi

As was presented in chapter 4.3 on ANSI C Stack interface, many Stack types include data in an unusable binary format and should be converted to formats that use types more suitable in Delphi. Types containing unusable formats should be named with prefix "OpcUa_" that would come after the 'T' letter, which is a common Delphi naming convention for classes, records and enumerations. Using the "OpcUa_" prefix is also the ANSI C Stack naming convention, thus naming these types in Delphi with a similar prefix maps them to the ANSI C types. Then, the well usable Delphi types would be named starting with the prefix "TPs", associating them to be part of a Prosys product.

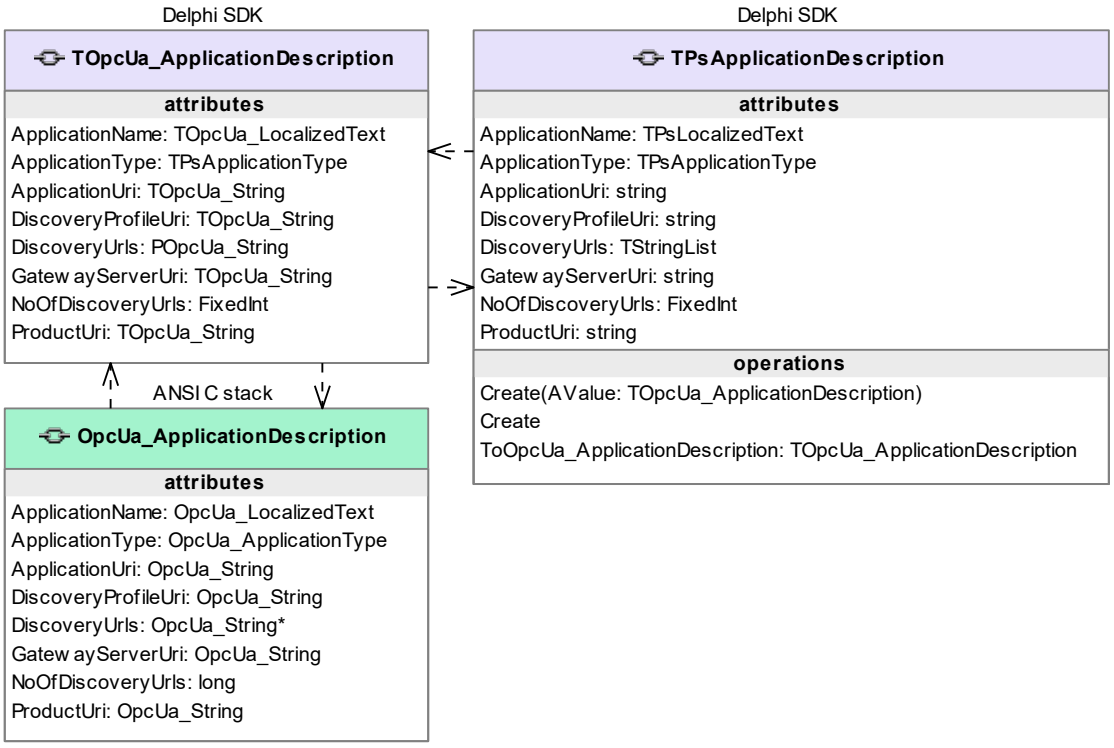

Figure 12: ApplicationDescription type definitions.

Figure 12 presents the needed code definitions of the ApplicationDescription type which is defined in the OPC UA Specifications [26]. *OpcUa_ApplicationDescription* is a Stack type and *TOpcUa_ApplicationDescription* its equivalent Delphi record whose data structures match exactly with each other. Therefore, they can be used in communication between the SDK and the Stack. As can be observed, the *NoOfDiscoveryUrls* field maps directly between C long and Delphi FixedInt types,

as does the *ApplicationType* field since it is an enumeration. The rest of the fields are of types containing binary string information and cannot be handled effectively in Delphi.

This is solved with the *TPsApplicationDescription* class that contains only easily usable Delphi types. It has two constructors: one to create an uninitialized class, as was proposed in section 3.3.2, and another one to initialize an instance with a *TOpcUa_ApplicationDescription* record converting all the members from unusable types to more suitable types. To use an instance of this class as a parameter to the Stack, it needs another convertion method named *ToOpcUa_ApplicationDescription* that returns a corresponding *TOpcUa_ApplicationDescription* record.

The presented pattern is the proposed design for all types containing data in unusable formats. It includes defining a Delphi class that contains only standard Delphi types. An object of such a class can be created by initializing it a with a record that is equivalent to a Stack type, and the object should be able to be converted back to a record type.

5.3 Use case 1: Connecting securely to a server

5.3.1 Connection and initialization API

The requirements defined in section 4.2.1 were used as the basis for performing the connect use case, after which the API design guidelines presented in chapter 3.3 were applied to create the design.

Figure 13 illustrates the needed steps for initializing a *TPsUaClient* object and connecting securely to a server. The most notable issue that emerged when converting the connect use case API to Delphi was with memory handling, which was discussed in section 3.1.7. With Java, an application developer creates instances of *UaClient*, *ApplicationDescription*, *ApplicationIdentity* and *UserIdentity* classes. Thinking from a user perspective, as was the proposed approach in section 3.3.1, this works appropriately in Java since the user does not need to care for the objects anymore, all the created objects will be freed automatically. However, a Delphi user would need to free the equivalent objects properly when they are not needed anymore. This is not an issue for the *TPsUaClient* object, which is needed for all the client functionality in any case, but creating and freeing also the other three objects clearly adds work for the user. Hence, to make the initialization phase more usable, it was modified so that those three objects are created in the constructor of *TPsUaClient*

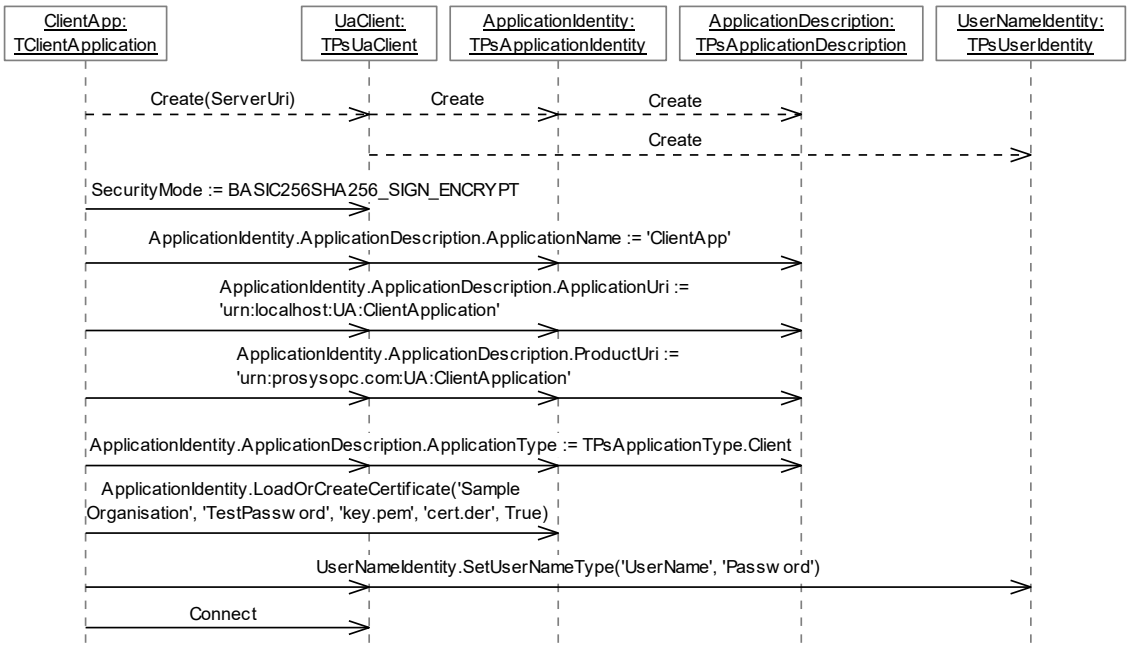

Figure 13: Sequence diagram of the connect use case

and freed in its destructor. The user only needs to set the equivalent parameters.

Furthermore, in the Java implementation these objects in *UaClient* were accessed with getter and setter methods, but as was proposed in section 3.1.4, they are converted to properties in the Delphi version. Also the Security Mode variable is converted into a property.

Interestingly, these modifications solve two other issues. They make calling the *LoadOrCreateCertificate* method simpler since the *TPsApplicationDescription* object is not required as a parameter anymore. As was discussed in section 3.3.4, this should improve the usability of the process. Also, as was proposed in section 3.3.3, factory methods should not be the favored approach in an API. In the Java SDK, the factory method *LoadOrCreateCertificate* is used to create an *ApplicationIdentity* object. In the Delphi version, all the objects are now already created and available for the user. Instead of needing to seek for a proper constructor, there is no need to seek for a constructor at all, and the *LoadOrCreateCertificate* method in Delphi is used more like a setter method.

In addition to removing the ApplicationDescription parameter, calling the *LoadOrCreateCertificate* method is slightly changed in comparison to the Java version. In Java, it gets the Private Key and Certificate parameters as *File* objects. Again, to make object handling in the Delphi version easier, strings containing locations of

these files are used instead of objects that would need to be freed later.

Finally, setting the user type is also changed. Since an application developer is no longer required to create a *TPsUserIdentity* object, the *TPsUserIdentity* class is provided with a setter method that can be used to set the type as a UserName type, which also requires a user name and a password.

5.3.2 Performing the connect use case

Figure 13, presenting the connect use case API, is examined to find out the steps the Delphi SDK needs to perform. First, a *TPsUaClient* object is created. In the equivalent *UaClient* class of the Java SDK, various fields such as security settings are initialized with their default values and these can be implemented similarly in the Delphi version. Default instances of the mentioned *ApplicationDescription*, *ApplicationIdentity* and *UserIdentity* classes are also created in Java and this can be implemented similarly in Delphi as well.

ApplicationDescription is a data type defined in the OPC UA specifications, describing an application with various parameters [26]. Its Delphi implementation follows the Stack type design. Then, *LoadOrCreateCertificate* is used to initialize an *TPsApplicationIdentity* object with a certificate. The method performs a complex process, which is examined in detail in section 5.3.3.

The *TPsUserIdentity* class can be translated from its Java equivalent with two modifications. As was mentioned in the previous section, setter methods should be implemented to provide an application developer the possibility to change the user type without having to create a new instance each time a user type is set. Also, the class needs to be capable of handling Certificates for the Certificate User Type. This functionality is implemented differently in the ANSI C and Java Stacks, requiring an architectural change for the Delphi SDK presented in section 5.1. Finally, connecting to a server is studied in section 5.3.4.

5.3.3 The *LoadOrCreateCertificate* method

LoadOrCreateCertificate, depicted in figure 14, is used to initialize a *TPsApplicationIdentity* instance with OPC UA identification information types ApplicationDescription and ApplicationInstanceCertificate. First, the method *ValidateApplicationDescription* checks the *ApplicationUri* and *ApplicationName* parameters of the *TPsApplicationDescription* instance. In case the parameters contain the string

"localhost", specifying the server should be on the same host, "localhost" is replaced with the actual hostname of the host machine.

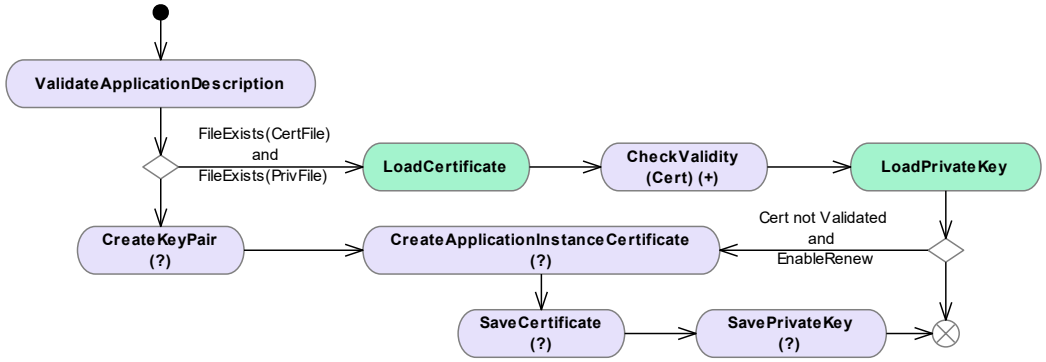

Figure 14: *LoadOrCreateCertificate* activity diagram.

If Certificate and Private Key files are found, the Certificate file is first loaded using a Stack function *LoadCertificate*. This functionality is provided differently in the Java and Delphi Stacks as was examined in section 5.1 on client architecture. Then, the *CheckValidity* method is used to check the validation time of the Certificate. In the Java SDK, a method with this name is called in a package provided by the Java framework. There is no such method available for Delphi, hence a tailored method for the task should be implemented using the OpenSSL library presented in section 3.4.4. The *LoadPrivateKey* method loads a Private Key from a file and can be implemented similarly to the *LoadCertificate* method.

In case the Certificate is not validated but should be renewed, specified by the parameter *EnableRenew*, the Certificate Public and Private Keys are used to create a new Certificate. In the Java implementation, this is performed by calling a Stack method *CreateApplicationInstanceCertificate*. An equivalent function named *OpcUa_Crypto_CreateCertificate* exists in the ANSI C Stack, but studying the sample programs shows the function is never called, implying it is probably not fully functional. The function should be tested, and in case it does not work, a similar one would need to be implemented by hand.

If either the Certificate or PrivateKey file is not found, a new KeyPair containing both the Public and Private Key is created for which the Java SDK uses a Stack function *CreateKeyPair*. Also in this case, a similar function named *OpcUa_Crypto_GenerateAsymmetricKeypair* exists in the ANSI C Stack but is never used. Thus, the ANSI C Stack function should be tested, and if it is not

functional, a tailored method should be implemented in Delphi.

The methods *SaveCertificate* and *SavePrivateKey* save the Certificate and the PrivateKey to their corresponding files. In the Java implementation they exist and are called in the Stack, and while the ANSI C Stack again contains equivalent functions, they are never called proposing the need to test them and possibly to implement them by hand.

5.3.4 The *Connect* method

The *Connect* method of the *TPsUaClient* class performs the actual connection process and is depicted in figure 15. Examining the first step of the connection process presents an architectural difference between the Java and ANSI C Stacks. In the Java SDK, a function named *InitClient* is called to initialize a Java Stack *Client* object. It is given a few parameters, such as an Application Instance Certificate, and is later used to access various client methods. In turn, the ANSI C Stack does not implement a corresponding client structure on the interface, instead all the functions are usable already after initializing the Stack at a platform layer, while the equivalent parameters are set in different stages of the connection process. Thus, *InitClient* should be dropped in the Delphi SDK.

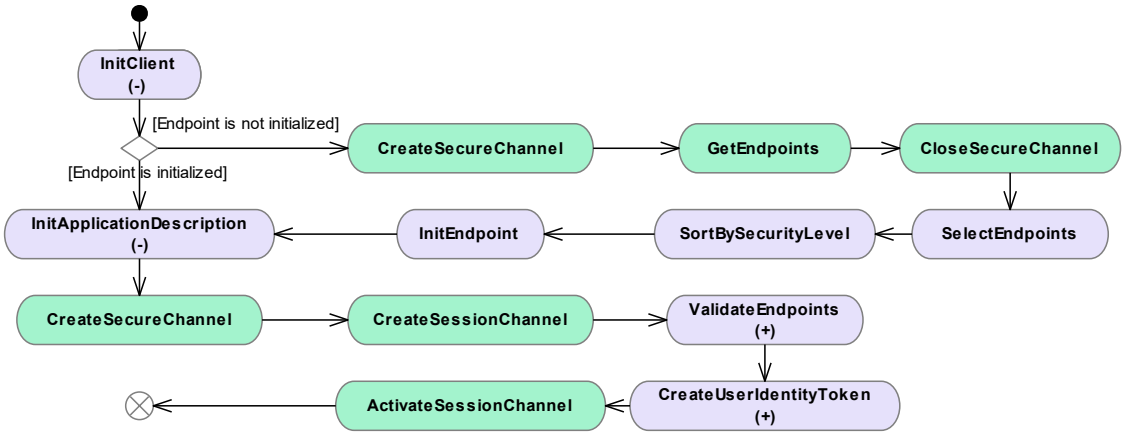

Figure 15: Activity diagram of the *Connect* method.

An Endpoint is a configuration of security settings and protocol, and it is used to present one possible connection configuration for connecting to a server [21]. When the Endpoint to connect to is not initialized, an automated process is performed to find an Endpoint that matches with configured connection settings. First, Security Mode is temporarily set to None and a Secure Channel is created with

the *CreateSecureChannel* method to a server to use the Discovery Services (2.1.2), which are available without creating a Session. The available server Endpoints are retrieved using the *GetEndpoints* service, after which the Secure Channel is closed with *CloseSecureChannel*. Then, of the available Endpoints, the *SelectEndpoints* method selects the ones that match with user-configured settings: a server URL, transport protocol, Security Mode, Security Policy and a Server Application Instance Certificate can be used as the filters. If no Endpoint is a match and the URL is set as localhost, the host name is used as an URL to select the endpoints, and if none are still found, matching the URL is discarded. The selected Endpoints are then sorted by their security level in *SortBySecurityLevel*, and the Endpoint with the strongest security level is chosen in the *InitEndpoint* method. As can be seen in the figure by observing the lack of ?, - and + marks on these methods, the Endpoint initialization functionality can be implemented very similarly in both SDKs.

In the Java SDK, the *InitApplicationDescription* method is used to set the *ApplicationDescription* instance of the Java Stack *client* object, but this method should be dropped in the Delphi implementation since there is no equivalent ANSI C Stack client structure. Also, a *TPsApplicationDescription* instance is attached later to the ANSI C Stack when a Session is created.

After having the Endpoint initialized, the *CreateSecureChannel* method is called to create a Secure Channel according to the Security Mode. If it is set to something else than None, the client Application Instance Certificate and the Private Key are provided to the server as well as the server Application Instance Certificate retrieved with *GetEndpoints*. Next, a Session is created by calling method *CreateSessionChannel*.

Next, *ValidateEndpoints* validates that an Endpoint received in the *CreateSessionChannel* method matches with an Endpoint received in the *GetEndpoints* method to verify the server is still the same, according to the OPC UA Specifications [26]. In the Java implementation, this is performed in the Stack, but as there is no equivalent function in the ANSI C Stack one should be implemented as a Delphi SDK method.

Then, four versions of *CreateUserIdentityToken* methods are used to create a specific User Identity Token according to the chosen User Type that were presented in section 2.1.4. In the Java SDK, these methods exist in the Stack. For the Delphi SDK, they need to be implemented by hand.

5.4 Use case 2: Browsing an Address Space

The Java API of the browse use case can be converted directly to Delphi, resulting in an API that is presented in figure 16. In this case, the *TPsIdentifiers* class stays as a factory class as opposed to what was suggested in section 3.3.3 on factory constructors. The reason is that this way the *TPsIdentifiers* class offers predefined NodeIds as its methods, which makes finding a needed NodeId easy. In the use case, a user also creates a *TPsNodeId* instance and retrieves a list of *TPsReferenceDescriptions* instances, after which the user is required to free them afterwards. This is considered appropriate, because this is information that the user will need for further purposes.

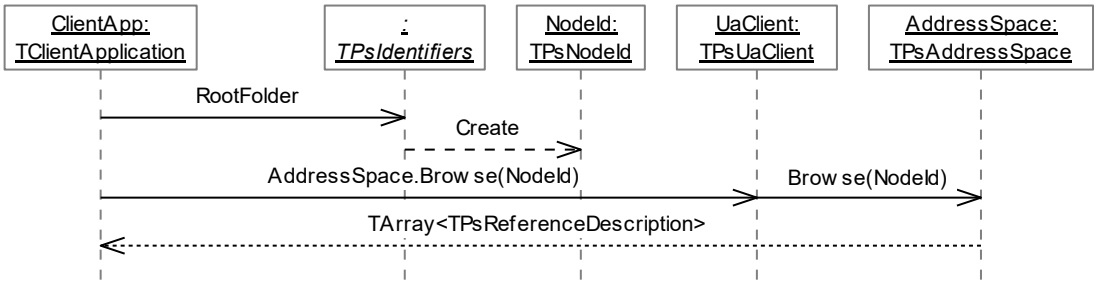

Figure 16: Sequence diagram of the browse use case API.

First, the Node to browse is identified with an instance of class *TPsNodeId* which is the Delphi SDK equivalent to the OPC UA type NodeId [25]. Thus, implementing the *TPsNodeId* class follows the design in section 5.2. Next, creating a *TPsNodeId* instance can be implemented as in the Java SDK. This requires converting the static methods of the Java SDK class *Identifier* into class methods for the corresponding Delphi SDK class *TPsIdentifiers*, as was discussed in section 3.1.6. Also, in the Java SDK, the *Identifiers* class is a Stack side class. There is no similar structure in the ANSI C Stack, thus, the *TPsIdentifiers* class needs to be implemented by hand for the Delphi SDK.

The *Browse* function is accessed through the other main class of the client SDK, *TPsAddressSpace*. It has an instance in the *TPsUaClient* class as a property. A *TPsNodeId* instance is given as a parameter, after which the OPC UA Browse service can be requested. This requires RequestHeader parameters [26], which are members of classes *TPsUaClient* in the Delphi SDK and *UaClient* in the Java SDK. Therefore, a circular reference occurs between the *TPsUaClient* and the *TPsAddressSpace* classes, but this can be solved by creating a reference to an interface of the *TPsUaClient* class as was proposed in section 3.1.2. After calling the service,

an array of *TPsReferenceDescription* instances is returned. ReferenceDescription is a type defined in the OCP UA specifications providing information on how a Node is connected to its neighbouring Nodes [26]. Again, the corresponding *TPsReferenceDescription* class can be implemented following the design proposed in section 5.2.

5.5 Use case 3: Reading and writing Node values

The Delphi API for reading and writing Node values is illustrated in figure 17, and as in the previous use case, it can be translated directly from the corresponding Java API. The *ReadValue* method reads the value of a Node specified by an instance of *TPsNodeId*. The return value is an instance of class *TPsDataValue* that can be implemented in the fashion proposed in section 5.2, as it is the Delphi equivalent to the OPC UA type DataValue [26]. The *WriteValue* method provides access for writing a value to a Node specified by NodeId. The value is specified by the second parameter, in this case it is 10. The method returns true if the write was performed, and false in case the writing will happen asynchronously later. The write and read operations are performed using the Write and Read Services [26].

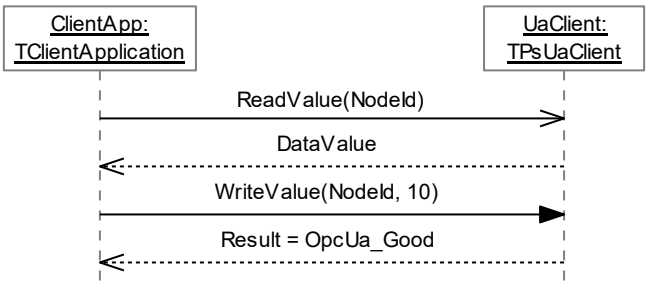

Figure 17: Sequence diagram of the read and write Node values API.

5.6 Use case 4: Subscribing to a Node

5.6.1 Initializing a Subscription

Figure 18 depicts the Delphi SDK API when subscribing to a Node. A notable modification was made in comparison to the Java version when the listener practice was changed to Delphi events as was proposed earlier in section 3.1.10. This resulted in the omission of the corresponding Delphi *SubscriptionNotificationListener* and

MonitoredDataItemListener interfaces. Instead, to be notified of data changes, a Delphi SDK user associates customised methods to the *OnDataChange* properties of the *TPsMonitoredDataItem* or *TPsSubscription* classes. Except for this, the API is converted directly from the Java SDK. The user needs to create and free the needed *TPsSubscription* and *TPsMonitoredDataItem* instances, but this considered acceptable because most times the "user" is expected to be the RAD layer, discussed in section 2.2.2, where created objects are freed automatically. Finally, in this example, the methods *HandleItemDataChange* and *HandleSubscriptionChange*, tailored by the user, are called each time the data value changes. Normally, a user would not want to attach methods to both the *TPsSubscription* and *TPsMonitoredDataitem* objects, but in this use case both are shown to present the different options. The value parameters are of type *TPsDataValue* implemented earlier in section 5.5.

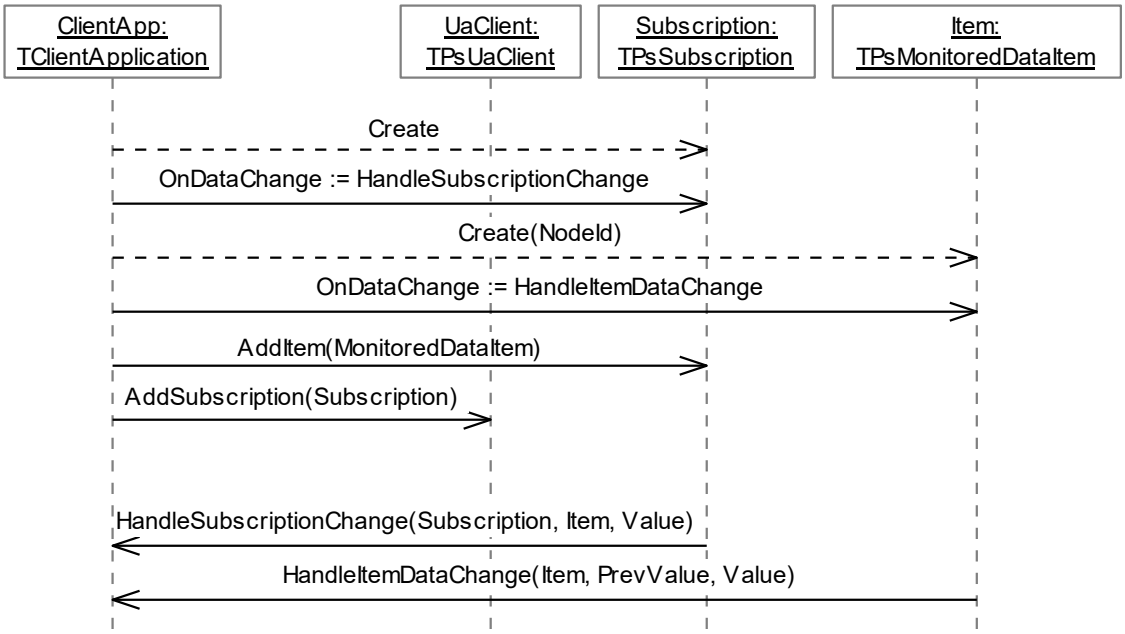

Figure 18: Sequence diagram of the subscribe use case API.

As to the implementation, the classes not presented in the previous sections, *TPsSubscription* and *TPsMonitoredDataItem*, are typical SDK side classes that can be translated directly from their correspondent Java classes. However, to be notified of data changes requires a process more complex which is examined in the following section. Also, as can be noted by looking at the figure, all the methods are translatable directly by following their Java SDK implementations.

5.6.2 Publishing Requests and handling Publish Responses

In order to notify an application of data changes, an OPC UA client needs to Publish Request messages to an OPC UA server and handle Publish Responses received from the server [21]. These messages are also used to indicate that a client or a server has not timed out. Figure 19 presents the design for the continuous Subscription process from Publishing to reacting to data changes. In this use case, also the classes where the actions take place are depicted. The starting point is the method *SendPublishRequest* which is called constantly in an infinite loop in the *TPsPublishTask* class. In the method, Publish Requests are sent to an OPC UA server by calling a Stack function named *PublishAsync*. The *SendPublishRequest* method is being executed in its own thread so that it is not interrupted by other processes. All this Publish functionality can be translated directly from the Java SDK.

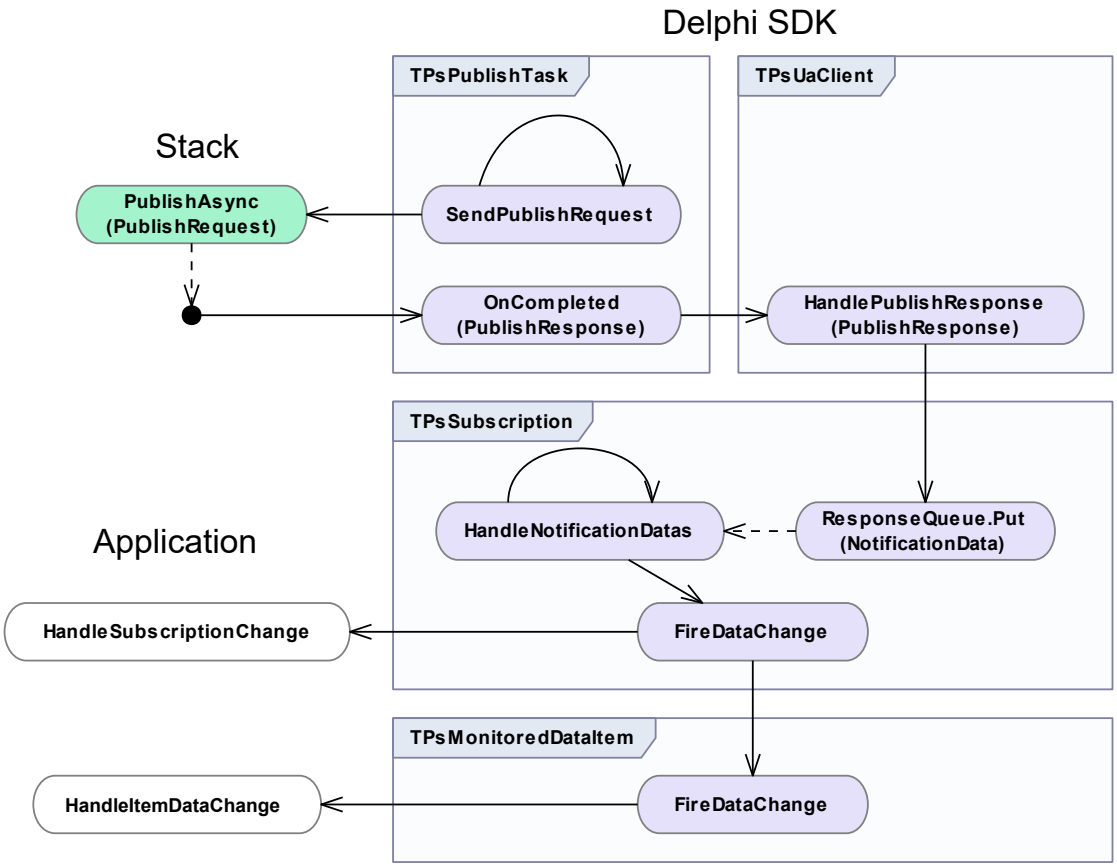

Figure 19: Publishing Requests and handling Publish Responses.

Next, the OPC UA server sends a Publish Response message to the client. This message goes from the Stack level to the *OnCompleted* methods in both the Java and Delphi SDK versions. However, there is an architectural difference as in the Java SDK the method is part of a specific transport interface whereas in Delphi it is specified with the ANSI C Stack as a callback method. After this, the Publish Response message is forwarded to the method *HandlePublishResponse* of *TPsUaClient* where it is first handled as a Keep Alive message so that the client knows the server has not timed out. The same message is then checked for possible Notifications indicating data values have changed. If there are any, the matching Delphi *TPsNotificationData* instance is then placed to a ResponseQueue of the *TPsSubscription* instance that is subscribed to the Nodes whose values were changed.

Again, there is an infinite loop checking for data to be handled. The *HandleNotificationDatas* method is checking for Notifications in its ResponseQueue. Whenever there are Notifications available, the method *FireDataChange* of *TPsSubscription* is called where the *OnDataChanged* property is checked for event handler methods attached by the application. In this case, the application has attached its method *HandleSubscriptionChange* to handle data value changes in a Subscription, and thus the method is called. Next, for each *TPsMonitoredItem* instance whose data value was changed, it is similarly checked if an application side method has been attached to look for data changes in a specific Node. Here, the application method *HandleItemDataChange* is attached and is then called to notify the application.

The needed container classes *TPsNotificationData*, *TPsDataChangeNotification*, *TPsPublishResponse* are based on types defined in the OPC UA Specifications [26]. Thus, they can be implemented as Stack types as is proposed in section 5.2.

6 Implementation

6.1 Initial code generation

The class architecture of the Java SDK had been modeled to a degree with ModelMaker (see 3.4.5). For a straightforward start in the implementation of the Delphi SDK, ModelMaker was used to generate the corresponding Delphi classes. This resulted in around 3000 code lines containing 30 classes with most of their methods and variables declared after a few hours of work. A Delphi SDK project was created with RAD Studio and the generated files were added to the project. However, ModelMaker was unable to link the units appropriately with each other, resulting in code with a huge number of cross-references and data types used in the Java SDK that were unrecognized in the Delphi project. Evidently, such a project that could not be compiled. For compiling the project, all cross-references between classes and references to unknown data types were first commented out so that after defining them they could be used again with a little effort.

6.2 Implementation of the use cases

The required use cases were implemented following the design in the previous chapter (see 5). During the implementation of the *LoadOrCreateCertificate* method (5.3.3), it was noted that the cryptographic utility functions in the ANSI C Stack that were not used in the sample applications did not work when they were tested. Thus, as was anticipated, the methods *CreateKeyPair*, *CreateApplicationInstanceCertificate*, *SaveCertificate* and *SavePrivateKey* needed to be implemented by hand using the OpenSSL library.

Debugging was performed on three levels. Mostly, the actual RAD Studio debugger was used to trace errors. As was planned in section 3.4.6, Prosys Simulation Server provided connection and debug logs that were used to find out issues related to OPC UA communication. To find out errors in the Stack layer, the Stack code was modified to write trace information to files. Close to every time, problems in the Stack level were a result of incorrect parameters given to Stack functions from the Delphi SDK.

6.3 Tests

6.3.1 Performing the use cases

A simple OPC UA client test application was developed in Delphi to test the Delphi SDK prototype. The test application was used continuously during the development of the Delphi SDK as a part of code debugging and to verify that the use cases could be performed as they were required to be performed. The test client was connected to a locally running Prosys Simulation Server, which confirmed that the Delphi implementation using the ANSI C Stack is able to connect to a software that uses the Java SDK on top of the Java Stack. All the required use cases were performed succesfully. More tests, especially with servers running on different Stack implementations, will be performed in the upcoming development phases to verify the interoperability of the Delphi SDK with the other Stack implementations, but it is not in the scope of this thesis.

6.3.2 Memory management

Along the implementation of the Delphi SDK, FastMM was used for detecting memory leaks and finding the sources of the memory leaks in the Delphi code. This way the leaks could be traced in the code and fixed, which consequently resulted in the Delphi SDK not leaking any memory, at least in the SDK level. Thus, as memory management is a crucial topic when evaluating how the Java code could be translated to Delphi, it can be concluded that the methods found were suitable and should be used in future too.

However, detection of memory leaks in the ANSI C Stack DLL memory is not possible using FastMM. Other methods and memory analyzers were tried and looked for to find a solution, but nothing as precise as FastMM was found. Also, if a memory leak is caused by a programming error in the Stack, it can not be fixed from the Delphi SDK. Most common memory leaks in the Stack level would be caused by the Delphi SDK not using the wrapper methods to free memory that has been allocated in the Stack when returning values from function calls. A method that can be used to evaluate the memory usage is opening the Windows Task Manager and looking at its Details page, where the amount of physical memory used by a process is shown. Then a specific method, for example the *Browse* method, can be called repeatedly to see if the memory usage of the test application is increasing, which would indicate that the *Browse* method leaks memory.

7 Conclusions

7.1 Answers to the research questions

The first research question was: *"How can the existing Prosys OPC UA Java SDK be translated to Delphi?"* In most cases, translating a single method was found to be very straightforward: the syntax rules, the basic data types and the naming conventions could be converted directly to their equivalent Delphi representations. Memory management issues needed to be handled more specifically and affected even the API design. Also differences between the Java and Delphi frameworks, e.g. finding the Delphi equivalent for a specific utility method or type, needed case-specific consideration and in some cases a hand-made implementation, such as with creating a Certificate. As a whole, the Delphi framework proved to provide good utilities for creating the Delphi SDK. In regards to the big picture, Delphi's object-oriented features were suitable for translating the Java SDK Client architecture almost 1-to-1 while the UML diagrams were used successfully to illustrate class structures and complex processes in a more comprehensible format than the source code.

The second research question was: *"How can the wrapping of the ANSI C Stack DLL be implemented?"* Using the Stack DLL needed its types and functions to be defined exactly as their equivalent Delphi types and methods, after which the functions could be called with Delphi. The Stack error codes could be translated to Delphi exceptions very conveniently, but the string type differences between Delphi and the Stack resulted in creation of an OPC UA data type layer, which again led to constant data type conversions.

The third research question was: *"How can the OPC UA Delphi SDK API be designed?"* First, the UML diagrams were used to present how OPC UA application use cases could be performed with the Java SDK. Next, the UML diagrams were modified to present the use cases in their respective Delphi forms. After this, the user-centric approach was applied as well as the API design guidelines and study results were used to modify the APIs for better usability. This was considered to be a good practice and provided practical development information, although to fully evaluate the application of this knowledge, more use cases than four would need to be implemented.

The fourth research question was: *"How serious and what type of disadvantages are there in developing the Delphi SDK based on the Java SDK and on top of the*

ANSI C Stack?" Clearly, the biggest source of inconvenience was dealing with a Stack written in a language other than Delphi, in this case ANSI C. It took time to adopt to the Stack function call conventions with correct parameter types and values, after which the data type conversions and memory management actions still had to be implemented close to every time a Stack function was called, resulting in lots of work and a remarkable number of code lines. The second biggest development issue was the memory management. In comparison to the garbage collector of Java, Delphi needs appropriate manual allocation and freeing of memory as does ANSI C. Furthermore, while the memory leaks in Delphi could be traced effortlessly with FastMM, it is close to impossible to guarantee that all the Stack memory is freed.

7.2 Future work

This thesis examined how the OPC UA Delphi SDK can be developed based upon Prosys OPC UA Java SDK. The methods found were used to develop the Delphi client SDK prototype which successfully performed the use cases as it was required to perform. Based on these observations, it can be concluded that suitable methods for the development process were found. In future, these methods will be used to develop the complete OPC UA Delphi SDK. However, since the prototype implements perhaps one fourth of the final SDK, most of the Java SDK code was not converted. There is a realistic possibility that some of the proposed methods are not applicable exactly as described in this thesis and that some remarkable issues did not occur during the implementation.

It was noted that the code using the ANSI C Stack had to be written manually, instead of translating it from the Java SDK which uses the remarkably different Java Stack. Then again, many methods that were not on the Stack interface level could be translated to Delphi very directly. On these occasions, a sort of primitive Java-to-Delphi compiler would definitely advance the translation process. Possibly, it could be capable of translating method definitions, basic data types, variable names, exceptions and the syntax. But implementing even such a compiler for only one project could be too laborious. If there was much more code than the Java SDK to be translated, the actual development should be given more consideration.

Also, a few possible further study topics emerge from this thesis. Even though lots of effort has been put on developing the Prosys OPC (UA) SDKs with good support and thinking of the customers, a complete API evaluation, including the cognitive

dimensions (see 3.3.1), of the Prosys SDKs could provide valuable information for future development. In regards to the Delphi SDK, it was noted that the interface between the ANSI C Stack required continuous data type conversions. The impact they have on performance has not been evaluated and continuous conversions of big data types could become an issue in performance-critical systems. These include mobile platforms that are expected to be used more commonly in industrial automation in future. Thus, also the performance of the Delphi SDK would be a valuable topic for further research.

References

[1] D. A. Plaisted. *Source-to-Source Translation and Software Engineering.* Journal of Software Engineering and Applications, vol.6, no. 4, pp. 30-40, 2013. DOI: 10.4236/jsea.2013.64A005.

[2] A. A. Terekhov, C. Verhoef. *The Realities of Language Conversions.* Journal of IEEE Software, vol. 17, no. 6, pp. 111-124, 2000. DOI: 10.1109/52.895180.

[3] MatrikonOPC. *MatrikonOPC OPC-UA Embedded Server Software Development Kit (SDK).* http://www.matrikonopc.com/opc-ua/embedded/sdk.aspx

[4] Softing. *OPC UA .NET / C# Server and Client Development Toolkit / SDK.* http://industrial.softing.com/en/products/software/opc-development-toolkits/opc-ua-net-development-toolkit/opc-ua-net-server-client-toolkit-for-windows.html

[5] Unified Automation. *Choose SDK.* https://www.unified-automation.com/products/sdk-overview/choose-sdk.html

[6] ModelMaker Tools. *ModelMaker: Native Refactoring and UML 2.0 modeling for Delphi and C#.* http://www.modelmakertools.com/modelmaker/index.html

[7] Embarcadero Technologies. *Extended Integer Types: ___int8, ___int16, ___int32, ___int64* http://docwiki.embarcadero.com/RADStudio/Berlin/en/Int8,_int16,_int32,_int64,_Unsigned_int64,_Extended_Integer_Types

[8] M. Reddy. *API Design for C++.* 1st edition. Elsevier, Inc. 2011.

[9] S. Clarke. *Measuring API Usability.* Dr. Dobb's Journal, vol. 29, pp. S6–S9, 2004. http://www.drdobbs.com/windows/measuring-api-usability/184405654

[10] J. Stylos, S. Clarke. *Usability implications of requiring parameters in objects' constructors.* Proceedings of the 29th international conference on Software Engineering. ICSE '07. IEEE Computer Society, pp. 529–539, 2007. DOI: 10.1109/ICSE.2007.92.

[11] M. Piccioni, C. A. Furia, B. Meyer. *An Empirical Study of API Usability.* IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 5-14, 2013. DOI: 10.1109/ESEM.2013.14.

[12] B. Ellis, J. Stylos, B. Myers. *The factory pattern in API design: A usability evaluation.* Proceedings of the 29th international conference on Software Engineering. ICSE '07. pp. 302-312, 2007. DOI: 10.1109/ICSE.2007.85.

[13] J. Bloch. *How to design a good API and why it matters.* In: OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications pp. 506-507, 2006. DOI: 10.1145/1176617.1176622.

[14] J. Bloch. *Effective Java.* 2nd edition. Addison-Wesley. 2008.

[15] T. Scheller, E. Kuehn. *Influencing Factors on the Usability of API Classes and Methods.* In: 19th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, 2012. DOI: 10.1109/ECBS.2012.27.

[16] J. Tulach. *Practical API Design: Confessions of a Java Framework Architect.* 1st edition. Apress, Inc. 2008.

[17] M. Henning. *API Design Matters.* Communications of the ACM, vol. 52, no. 5, pp. 46-56, May 2009. DOI: 10.1145/1506409.1506424.

[18] M. P. Robillard. *What Makes APIs Hard to Learn? Answers from Developers.* IEEE Software, vol. 26, no. 6, pp. 27-34, November/December 2009. DOI: 10.1109/MS.2009.193.

[19] R. Lischner. *Delphi in a Nutshell.* 1st edition. Sams O'Reilly & Associates, Inc, 2000.

[20] B. Eckel. *Thinking in Java.* 4th edition. President, MindView, Inc. 2006.

[21] W. Mahnke, S.-H. Leitner, and M. Damm. *OPC Unified Architecture.* Springer, 2009.

[22] Sa. Singh, Su. Singh, G. Singh. *Reusability of the Software.* International Journal of Computer Applications (0975 – 8887), vol. 7, no. 14, October 2010. DOI: 10.5120/1338-1703.

[23] W. B. Frakes and K. Kang. *Software Reuse Research: Status and Future*. IEEE Transactions on Software Engineering, vol. 31, no. 7, pp. 529-536, July 2005. DOI: 10.1109/TSE.2005.85.

[24] OPC Foundation. *OPC Unified Architecture Specification, Part 2: Security Model*. Release 1.03. 2015.

[25] OPC Foundation. *OPC Unified Architecture Specification, Part 3: Address Space Model*. Release 1.03. 2015.

[26] OPC Foundation. *OPC Unified Architecture Specification, Part 4: Services*. Release 1.03. 2015.

[27] B. Kernighan, D. Ritchie. *The C Programming Language*. 2nd edition. 1978.

[28] Embarcadero Technologies. *Using the RTL (Run-Time Library) - RAD Studio*. http://docwiki.embarcadero.com/RADStudio/Seattle/en/Using_the_RTL_(Run-Time_Library)

[29] Embarcadero Technologies. *Events (Delphi) - RAD Studio*. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Events_(Delphi)

[30] Embarcadero Technologies. *Delphi Data Types - RAD Studio* http://docwiki.embarcadero.com/RADStudio/Seattle/en/Delphi_Data_Types

[31] Embarcadero Technologies. *RAD Studio | Windows, Mac, Android and iOS App Development* https://www.embarcadero.com/products/rad-studio

[32] Microsoft Developer Network (MSDN). *Potential Errors Passing CRT Objects Across DLL Boundaries*. https://msdn.microsoft.com/en-us/library/ms235460.aspx

[33] Microsoft Developer Network (MSDN). *Allocating and freeing memory across module boundaries*. https://blogs.msdn.microsoft.com/oldnewthing/20060915-04/?p=29723/

[34] OpenSSL Software Foundation. *OpenSSL*. https://www.openssl.org/

[35] S. Alhir. *UML in a Nutshell*. O'Reilly & Associates, Inc. 1998.

[36] Prosys PMS Ltd. *About OPC and OPC UA* https://prosysopc.com/opc/

[37] Prosys PMS Ltd. *Prosys OPC UA Simulation Server*. https://www.prosysopc.com/products/opc-ua-simulation-server/

[38] Prosys PMS Ltd. *Prosys OPC SDK Sentrol*. https://www.prosysopc.com/products/opc-sdk-sentrol/

[39] Prosys PMS Ltd. *Prosys OPC UA Java SDK* https://www.prosysopc.com/products/opc-ua-java-sdk/

[40] B. Swart. *Using C DLLs with Delphi*. Dr. Bob's Delphi Clinic. http://www.drbob42.com/delphi/headconv.htm

[41] Delphi Bistro. *FastMM – Preparing your apps to report memory leaks*. http://delphibistro.com/?p=186